*Reference Manual*

# Assembler and Linker *for the Atari*

Six Forks Software • 11009 Harness Circle • Raleigh, NC 27614

# Table of Contents


Chapter 1 Introduction 1 ald1

Getting Started 1
Terminology 1
The Big Picture 1
The Linker 3

Chapter 2 The Distribution Diskette 5

Preparing the Assembler and Linker for Use 5

Chapter 3 Six Forks Assembler Language 6

Comparison with Atari's Assembler/Editor 6
The Source File 7
Statements 7
Numbers 8
Strings 8
Labels 8
VIRTUAL Labels 9
Restrictions on the Use of Virtuals 9
ENTRY labels 9
Reserving Object Program Memory 9
The Location Counter 10
Relocatable Code 10
Setting the Location Counter 10
The Initial Location Counter Value 11
Expressions 11
Attributes 11
The Current Location Counter Value 11
Expression Evaluation 12
Signed Versus Unsigned Values 13
Byte Selection on an Expression Result 13
Two-byte Operands 14
One-byte Operands 14
Automatic Selection of Zero-Page Instructions 15
VIRT8 Virtuals 15
The Comment Statement 15
The .BYTE Statement 15
The .CFE Statement 16
The .DBYTE Statement 16
The .ENTRY Statement 17
The .OUTPI Statement 17
The .VIRT8 Statement 17
The "=" Statement 17
The "*=" Statement 18
Instructions with no Operand Field 18
Relative Branch Instructions 18
JMP and JSR Instructions 19

Instructions with Several Addressing Modes 19

Chapter 4 Using the Assembler 21

Labelled Packs 21
Assembler Capacity 21
Pass 1 21
Pass 2A 22
Pass 2B 22
Starting an Assembly 22
Mid-Assembly Choices 22
Print Mode 22
View Mode 23
Error Reporting 23
The Assembly Listing 23

Chapter 5 Linker Concepts 26

The Linker Control File 26
Example 1: Illustrates all Linker Concepts 27
Example 2: Techniques to be Aware of 29

Chapter 6 Using the Linker 32

Linker Capacity 32
Instructions and Constants 32
The Internal Workarea 32
Starting a Build 33
Control File Reading 34
Error Handling 34
Input File Reading 34
Output File Building 34
The Load Map 34

Chapter 7 Multi-Group Programs 36

Memory Assignment for each Group 36
The Communication Vector 37
The .EOUT Statement 39
The Transparent Jump Statement 39
Executing a Multi-Group Program 41

Chapter 8 The Sample Program 42

Files on the Distribution Diskette 42
Assembling and Linking the Sample Program 42
Executing the Sample Program 43

Chapter 9 Programming Suggestions 45

Appendix A Error Codes 47

Appendix B Specifications 51

# Chapter 1

## Introduction

## Getting Started

ald24

We supply this manual and a diskette containing the software. You supply the things mentioned in Appendix B. We suggest that you get started by doing the following:

1. Copy the distribution diskette to a backup. Make boot packs as explained in chapter 2.

2. Assemble and run the sample program. Chapter 8 tells how to do that.

3. Read over the programming suggestions in chapter 9.

4. As a first programming effort using this package, you might make a modification to the sample program.

## Terminology

A SOURCE FILE is ASSEMBLED to produce a RELOCATABLE FILE. Relocatable files are combined by the linker to produce an OBJECT FILE. An object file is also called an EXECUTABLE FILE, an ABSOLUTE FILE, and a BINARY file.

The process of running the linker to create an object file is called a BUILD. Chapter 7 uses the term "GROUP" to denote an object file that is only part of a total object program.

A UNIT has two forms: a source file and its corresponding relocatable file. Thus, the term "UNIT" refers to a logical piece of a program without regard to its form. An ASSEMBLY is a unit whose source code is assembler language.

A relocatable file can contain both relocatable code and absolute code.

## The Big Picture

Our Assembler/Linker package provides a method of software development that allows a program to grow very large without becoming unmanageable. The method, which is based upon the linker, is the industry standard method for developing software. Commercially developed programs tend to be large for two reasons:

1. Computers, even small ones by today's standards, can contain large programs.

2. Small programs tend to grow, and for good reason. Building on top of what's already there is the most productive and profitable of programming activities.

These reasons are applicable to Atari users, whether programming for enjoyment or profit. Hardware capacity is ample. For example, the built-in operating system, which occupies only 8k of memory, is a 130-page assembler language program. At that rate, a 40k program would have over 600 pages of source code.

Don't think in terms of how large a program you can or want to develop. Just be aware that the most powerful and exciting programming ideas usually involve adding to what's already there, and that it is nothing less than a tragedy when this incremental growth process becomes impossible for reasons that could have been avoided.

Be aware also that with the proper programming strategy and development tools, a program can grow almost indefinitely in size without becoming unmanageable. Here's how it's done:

The program is constructed as a set of subroutines. The programmer is able to forget how a subroutine works and remember only how to use it. Detail is contained, and a program can grow without becoming less comprehensible. The source code is divided into multiple source files, each small enough to be conveniently edited and reassembled. Each source file contains a logical "unit" of software, typically a single subroutine or small group of them. An assembler language source file seldom contains more than 50 instructions. Complete with commentary, it is seldom longer than 4 or 5 pages.

To assist in debugging, the units often contain internal consistency checks. Data structures often contain "checkbytes" that can be verified by the units. With such measures, the object program can become to a great extent its own debugger.

Now, how can source code that is spread over many source files be converted into an executable program?

Let us first consider the single-stage process that is exemplified by Atari's present assembler products. This process translates source code directly to executable form. To allow for multiple sources, the assemblers typically allow multiple source files to be "included" in a single assembly. However, even if the assembler could handle the total amount of source code, the time to reassemble and the size of the resulting listing make this approach unworkable.

The other possibility with the single-stage process is to have each unit be a separate assembly, yielding a separate executable file. That eliminates the problems of a single large source, but it puts two housekeeping chores on the programmer's shoulders:

1. The object code for each unit must be assigned a place in memory. The exact place seldom matters as long as the units do not fall on top of each other.

2. The units must be linked together. For instance, a subroutine defined in one unit must be called from another unit.

If done manually, these chores become the most time consuming and discouraging part of the programming effort. The memory assignment for each unit must be done with an "absolute origin" statement in the source file. Thus, to move a unit, it must be edited and reassembled. Linkages between the units require that the source files contain absolute addresses of points within other units.

It is easy to see that even a minor logic change to a single source file can lead to much additional source code modification that is unrelated to the logic change. To postpone these labors, error prone shortcuts such as machine language patches are employed.

The software industry came to grips with these problems in the early 1960's. That was when the linker was developed.

## The Linker

Although unfamiliar to many home computer programmers, all major computer systems, including the mainframes, the IBM PC, Apple's Macintosh, UNIX-based, and CPM-based systems, have linker-based development packages.

The linker combines software units called "relocatable files" into an executable object program. A relocatable file, which is the result of assembling a source file, can be thought of as the source file's equivalent, but compacted. Commentary is removed. Instructions and data are represented in machine language form but not yet in executable form because relocation and handling of external labels have yet to be done by the linker.

### RELOCATION

A unit can be programmed without indicating where its object code is to go in memory. The linker assigns memory to these units so that they lie one after another with no wasted space in between. This process is called "relocation".

### EXTERNAL LABELS

They are the means for linking units together. Labels are defined in the source code in the normal manner. A label that is to be visible to other units is declared to be an ENTRY. When another unit references an ENTRY label, the linker places the proper absolute value into the reference.

For example, to define a subroutine in one unit and call it from another, the following is done in the source code:

1. The subroutine is programmed in the normal manner with a label designating its name (its entry point).

2. In the subroutine's source file, that label is declared to be an ENTRY.

3. Calling the subroutine from the other unit is done as though the subroutine were defined locally. In assembler language, the call consists of a JSR instruction with the subroutine's name in the operand field.

The linker processes our example as follows (assuming both units are made relocatable, which is normally the case):

1. It relocates the unit containing the subroutine. Thus, the absolute address of the subroutine's entry point is not known until the linker is run.

2. It relocates the unit containing the subroutine call.

3. It puts the absolute address of the subroutine into the operand field of the subroutine call.

In addition to creating the object file, the linker prints a "load map" showing each ENTRY label and its associated value. The load map is essential for debugging.

Although our assembler is currently the only language processor that feeds into the linker, the linker itself is language independent. A relocatable file could, for example, come from a Basic compiler. For those wishing to develop their own language processors, documentation of our relocatable file format will soon be available.

# Chapter 2

## The Distribution Diskette

It is an 810-compatible diskette that is not bootable because it does not contain DOS or DUP. It contains the following files:

ald25

SFASM The assembler, a load-and-run file.

SFLINK The linker, a load-and-run file.

PACKID The pack label file. Pack labels are explained in chapter 4. The distribution diskette's pack ID is ALPAK1. It is needed to assemble and link the sample program.

SSOUA-Q These 17 files are the sources for the sample program. See chapter 8.

LNKSP1 Control file for linking the sample program.

## Preparing the Assembler and Linker for Use

They can be executed directly from the distribution diskette using the "L" command, but that is a clumsy process since the distribution diskette is not bootable.

We recommend that a "boot pack" be created for each of the programs. Two diskettes are needed, or you can use the two sides of a single diskette. To make the assembler boot pack, do the following:

1. Format a pack and put DOS and DUP onto it.

2. Copy SFASM from the distribution diskette onto the boot pack.

3. On the boot pack, change the name of SFASM to AUTORUN.SYS.

Using SFLINK on the distribution diskette, make the linker boot pack. To start either program, simply boot the computer from the boot pack. Make sure no cartridge is inserted, and on the XL and XE models, hold down the Option key when booting.

# Chapter 3

## Six Forks Assembler Language

## Comparison with Atari's Assembler/Editor ald3

Our source language is modeled after that of Atari's Assembler/Editor. To assist those already familiar with Atari's language, here are the differences:

1. Our statements do not have line numbers.

2. We support ENTRYs and virtuals. ENTRY labels must be explicitly declared with the .ENTRY statement. Expressions containing virtuals have restrictions on their form.

3. We do not support the TITLE, PAGE, TAB, and IF statements. They are for making a large file more managable. When a linker is available, source files can be kept small. We also do not support the END statement.

4. We have an direct means of selecting the low or high byte of an expression result. For example, the statement "LDA #.LO.TABADR" does the same thing as "LDA TABADR-TABADR/256*256". Byte selection can be done on a virtual.

5. Using our .CFE declaration, a self-describing character constant can be defined.

6. We allow an immediate instruction operand to be stated as a one-byte string.

7. Statements that we support compare with Atari's as follows:

| STATEMENT | COMPARISON WITH ATARI | |
|---|---|---|
| instructions | identical | |
| comment | identical | |
| .BYTE | identical | |
| .CFE | not in Atari | |
| .DBYTE | identical | |
| .ENTRY | not in Atari | |
| .EOUT | not in Atari. | Explained in chapter 7. |
| .OUTPI | not in Atari | |
| .VIRT8 | not in Atari | |
| .WORD | identical | |
| *= | identical | |
| = | identical | |

## The Source File

It must be on a labelled pack. Pack labelling is explained in chapter 4.

The file name can be up to 8 bytes long. The first byte must be a letter. Remaining bytes can be letters or digits. An extension (an ".xxx") is not permitted.

The relocatable file created by the assembler is given the name of the source file with an extension of ".R". For example, if source file "SFILA" is assembled, the relocatable file is named "SFILA.R".

The source file is in the conventional DOS form. Each record ends with an EOL. Each record contains one source statement.

Records can be up to 130 bytes long, but they are normally kept short enough to be contained in one line of the assembly listing.

There is no special "last" statement such as an END statement. The END statement is not supported.

Following is a sample source file, shown as it is likely to be entered:

```
; THIS IS OUR SAMPLE PROGRAM
;
 .OUTPI RELOC1
;
 .ENTRY ABC

ABC LDA #2 ENTRY POINT.
 STA XYZ+1
;
 RTS RETURN TO CALLER.
```

The remainder of this chapter is in two parts. First, the elements of the language are described. Then, each statement is described individually.

## Statements

If a statement begins with a semicolon, it is a comment. Otherwise, the statement is organized into four fields: label, operation, operand and comment.

The fields are separated by one or more blanks. To signify a label field, start it in the first record byte. If the first byte is blank, that means there is no label field, and the first nonblank byte is the start of the operation field.

Most statements have an operand field. Following it is the comment field, which is always optional. In the statements that have no operand field (e.g. TXA), any material following the operation field is treated as commentary.

Here is the assembly listing that would be produced by the above example:

```
                        ; THIS IS OUR SAMPLE PROGRAM
                        ;
                                .OUTPI RELOC1
                        ;
                                .ENTRY ABC
                        ;
0000R A902              ABC     LDA    #2          ENTRY POINT.
0002R 8DVVVV                    STA    XYZ+1
                        ;
0005R 60                        RTS    RETURN TO CALLER.
```

The fields are tabbed to fixed columns to improve readability. The only statement that must always be present is the .OUTPI statement. It tells the assembler where to write the relocatable file. The assembly listing is fully described in chapter 4.

## Numbers

A number can be given in decimal or hex form. Hex is indicated by beginning the number with "$". In addition to the decimal digits, hex numbers can contain the letters "A" through "F".

The value of a number can range from 0 through 65535 in decimal, or 0 through $FFFF in hex.

## Strings

A string is a sequence of characters enclosed in quotes. The sequence cannot contain the quote character used to enclose the string. Both the single and double quote can be used to enclose a string. The maximum string length is 100 bytes. Following are examples:

```
"THIS IS A STRING"
'THIS IS ALSO A STRING'
"DON'T TYPE TOO FAST"
""      (empty string, usable in .CFE statement)
```

## Labels

A label is up to 6 bytes long. It begins with a letter. Remaining bytes, if any, are letters or digits. A label cannot be "A" because that notation is used in the shift and rotate instructions to denote the A-register.

A label denotes a 16-bit value. A label is defined (assigned a value) when it appears in the label field of a statement.

## VIRTUAL Labels

When a label is used in a given source file but not defined in that source file, it is said to be a "virtual" in that source file. In the above example, XYZ is a virtual.

When a statement contains a virtual, the assembler cannot translate it completely to machine language because it does not know the value of the virtual. These virtual references are encoded into the relocatable file and are "resolved" (converted to machine language) by the linker.

## Restrictions on the Use of Virtuals

A label, whether it is locally defined or a virtual, is used in the operand field, where it is part of an expression (described later). Compared with locally defined labels, virtuals have the following restrictions on their use:

1. The expression containing a virtual must reduce at assembly time to "virtual plus or minus a constant". The constant value can be any 16-bit value.

2. A virtual cannot be used in the operand field of the "*=" statement. In other words, the location counter cannot be set to an expression containing a virtual.

3. A virtual cannot be used in the operand field of the "=" statement. In other words, a local label cannot be "equated" to a virtual.

## ENTRY Labels

Any label that is defined in a given source can be declared to be an ENTRY in that source. That makes the label and its associated value visible to the linker and capable of resolving virtuals in other assemblies.

Declaring a label to be an ENTRY in a given source has no effect on its use within that source.

## Reserving Object Program Memory

ald5

Most statements reserve object program memory. For instance,

```
LDA  #2
```

generates a 2-byte instruction. Two bytes of memory are reserved, and the machine language form of the instruction is placed in them. The statement

```
*=  *+20
```

reserves 20 bytes of memory, but does not place data into the bytes. Memory reservations always occur at the "next available byte".

## The Location Counter

The assembler keeps track of the "next available byte" with its location counter. A block of "x" bytes of memory is reserved as follows:

a. The current location counter value is the address of the first byte of the reserved block.

b. The location counter is incremented by "x".

In the assembly listing, the address field printed at the left of each memory-reserving statement is the location counter value in effect when that statement was encountered.

## Relocatable Code

The location counter can be absolute or relocatable. When an absolute location counter is in effect, memory is reserved at the indicated absolute memory locations.

When a relocatable location counter is in effect, memory is reserved at locations relative to the "load point" of the assembly, which is not known until the linker relocates it.

## Setting the Location Counter

The programmer sets it, and thereby controls where memory is reserved. It is set with the "*=" statement. Following are examples:

| | | | |
|---|---|---|---|
| | *= | $4000 | The location counter is set to an absolute address. Subsequent memory-reserving statements (until another "*=" is encountered) reserve memory in consecutive, ascending bytes beginning at location $4000. |
| XX | *= | ABSADR | Assuming ABSADR denotes an absolute address, the location counter is set to that address. The definition of ABSADR must precede the "*=" statement in the source file. The XX label is defined to denote the location counter value just before it is set to the new value. |
| | *= | RELADR | Assuming RELADR denotes a relocatable address, the location counter is set to that address. The RELADR definition must precede this statement. |
| ARRAY | *= | *+20 | The location counter is set to its former value plus 20. The effect of such incrementing of |

the location counter is to reserve memory without initializing it. The ARRAY label is defined just as was XX earlier. Thus, we see that ARRAY denotes the address of the first byte of the 20-byte area.

## The Initial Location Counter Value

The assembler initializes it to "0, relocatable". With that initial value, the linker relocates memory-reserving statements so that they fall immediately after the previous assembly.

When possible, an assembly should contain no location counter settings except for the "*=*+x" variety that reserves a block of memory. Such assemblies are the simplest to incorporate into programs because they are fully relocatable and occupy only one contiguous chunk of memory.

## Expressions

ald4

An expression is used whenever a numeric value is called for in an operand field. The expression is made up of one or more "terms". A term is a label, a number, a 1-byte string, or the current location counter value (indicated by "*").

Terms can be combined by addition, subtraction, multiplication or division. Either the low or high byte of the result can be selected.

An expression value is a two-byte (16-bit) quantity. When a one-byte value is called for, as in the immediate operand of an instruction, the high order byte of the expression result must be zero, which is to say that the value of the expression must be between 0 and 255.

## Attributes

Within the assembler, a term or expression result has one of three attributes:

| | |
|---|---|
| Absolute | The absolute value (a number between 0 and $FFFF) is known at assembly time. |
| Relocatable | The value known to the assembler (also between 0 and $FFFF) is relative to the "load address" of the assembly. |
| Virtual | Since a virtual label is present, the value cannot be computed by the assembler. The linker computes the final value. Expressions containing a virtual are also known as "virtual references". |

## The Current Location Counter Value

The symbol "*" denotes the location counter value that is in effect when the assembler starts processing a statement containing an "*". That value can be absolute or relocatable.

## Expression Evaluation

Expression evaluation is carried out in a simple left-to-right manner. To some users this might seem peculiar, as it differs from the standard practice of doing multiplications and divisions before additions and subtractions, but it is commonly used in assemblers including those from Atari.

When terms other than absolute ones are combined, there are restrictions on the arithmetic operations that are permitted. To explain the restrictions, we first look at how expressions are evaluated by the assembler.

Expression evaluation is centered around an "accumulator". As the expression is evaluated from left to right, each term is combined into the accumulator according to the operator that precedes it. For example, the expression

```
LABA*2+LABR
```

where

```
LABA = $24, absolute
LABR = $11, relocatable
```

is evaluated in the following steps:

1. The accumulator is initialized to "0, absolute".
2. LABA is added into it. The accumulator becomes "$24, absolute".
3. The accumulator is multiplied by "2, absolute". It becomes "$48, absolute".
4. LABR is added into the accumulator. This is a valid operation because a relocatable value can be added to an absolute one. The accumulator becomes "$59, relocatable".

The rules governing expression evaluation are:

1. The accumulator is initialized to "0, absolute".
2. An expression can begin with a minus sign. An absolute term must follow the minus sign. It is subtracted from the initial accumulator.
3. If the accumulator is absolute, the next operator and term can have the following forms:
   a. If the next term is absolute, then any of the four arithmetic operations are permitted. The accumulator remains absolute.

b. If the next term is relocatable, the operator must be "+". The accumulator becomes relocatable.

c. If the next term is virtual, the operator must be "+". The accumulator becomes virtual.

4. If the accumulator is relocatable, two forms of the next operator and term are permitted:

   a. The next operator is "+" or "-" and the next term is absolute. The accumulator remains relocatable.

   b. The next operator is "-" and the next term is relocatable. Here we are taking the difference between two relocatable values. The accumulator becomes absolute.

5. If the accumulator is virtual, the next operator must be "+" or "-" and the next term must be absolute. The accumulator remains virtual.

6. Division by zero results in an error message. Overflows from other operations are ignored.

## Signed Versus Unsigned Values

Terms and expression results are 16-bit values. Whether a given value is a signed or unsigned quantity is largely a matter of user interpretation. For example, $FFFF can also be thought of as -1 or as 65535, or as any other pattern of bits.

Addition, subtraction and multiplication operations yield "correct" results regardless of the interpretation, although you must be aware of overflows. There are two operations where a single result cannot satisfy both interpretations. In these operations, the assembler supports the unsigned interpretation. The operations are:

1. Division. Fortunately, division is infrequently used in assembler statements. In any event, division consists of a simple, integer, unsigned, 16-bit divide. We do exactly what Atari does.

2. Overflow check on expression results that must fit into one byte. The assembler and linker require that the high order byte of the expression result be zero. This interpretation prohibits negative values that can be correctly represented in one byte. For instance, the statement "ABC .BYTE -1" is unacceptable. As you will see, though, the statement "ABC .BYTE .LO.-1" is acceptable.

## Byte Selection on an Expression Result

An expression can begin with a "byte selection" prefix. Examples are:

```
.LO.LABA+2
```

```
.HI.LABV
```

This prefix is syntactically part of the expression. To aid in explaining its effect, we call the portion of the expression following the prefix the "inner expression".

There are no restrictions on use of the prefix. The inner expression can be absolute, relocatable, or virtual. When relocatable or virtual, the final result is determined by the linkage editor.

The attribute of an expression is not changed by adding the prefix. Byte selection is applied to the result of the inner expression in the following way:

.LO. The high order byte of the result is zeroed.

.HI. The value in the high order byte is moved to the low order byte and then the high order byte zeroed.

Byte selection is useful in setting up addresses at execution time. For example, to put the address "LAB1+1" into the A and Y registers, the following can be used:

```
LDA  #.LO.LAB1+1     A = LOW ORDER BYTE.
LDY  #.HI.LAB1+1     Y = HIGH ORDER BYTE.
```

## Two-byte Operands

ald6

An instruction that contains an absolute memory address has a two-byte operand. Also, a statement that defines a two-byte numeric constant has a two-byte operand. Here are examples:

```
JMP    LOOP
LSR    ACCUM+2
BIT    FLAG
LDY    ELEMENT,X
.WORD  XYZ+1          (LO,HI) ORDER.
.DBYTE XYZ-22         (HI,LO) ORDER.
```

## One-byte Operands

An immediate instruction has a one-byte operand. The zero-page form of an instruction has a one-byte operand. The statement that defines a one-byte numeric constant has a one-byte operand. Following are examples:

```
LDA    #47          IMMEDIATE INSTRUCTION.
LDA    $17          ZERO-PAGE FORM OF INSTRUCTION
.BYTE  PAGSIZ-4     ONE-BYTE NUMERIC CONSTANT.
```

Note: the BYTE statement is also used to define character constants.

The value of an expression used in a one-byte operand must lie between 0 and

255 as explained earlier. Violations of this rule are reported as soon as the final value of the expression is known.

## Automatic Selection of Zero-Page Instructions

Certain instructions have both an absolute-address form and a zero-page form. In that case, the assembler automatically chooses the zero-page form if the expression in the operand field has any of the following properties:

1. It yields an absolute value during pass 1 of the assembly process, and that value is between 0 and 255. Chapter 4 explains "pass 1".

2. It has a byte selection prefix (.LO. or .HI.).

3. It contains a VIRT8 virtual.

## VIRT8 Virtuals

A VIRT8 virtual is a virtual whose name has been used in a .VIRT8 declaration statement. The assembler makes the following assumption about VIRT8 virtuals:

> any expression containing a VIRT8 virtual will have a result that is between 0 and 255.

The only effect of a VIRT8 virtual is to cause the zero-page form of an instruction to be generated when that form exists.

It is your responsibility to see that an expression containing a VIRT8 virtual yields a result between 0 and 255. If an out-of-range result occurs, the linker will report the error.

The usefulness of the VIRT8 feature is that it allows zero-page locations to be referenced symbolically using virtual labels. Otherwise, each source file referencing a zero-page location would have to contain the absolute address of that location.

## The Comment Statement

ald7

If the first character in a statement is a semicolon then it is a comment. A blank line is also treated as a comment. It appears as a blank line in the assembly listing.

## The .BYTE Statement

Following are examples showing all forms:

```
TEN     .BYTE  10
XYZ     .BYTE  10,20,$30,.HI.A-2
HEAD    .BYTE  "THIS IS A HEADING"
```

```
MES1   .BYTE  'THE MAN SAID "I AM HERE"'
MES2   .BYTE  "DON'T SHARPEN THE PENCIL"
STUFF  .BYTE  2,"ABCD",3,"EFGH"
```

One or more subfields are given with a comma (but no blanks) between each. Each subfield is either an expression or a character string. Expression values must lie between 0 and 255. If the expression contains a virtual, the final value is computed and checked by the linker.

## The .CFE Statement

It creates a self-describing character constant. The operand field consists of a single character string. Following are examples:

```
TITLE  .CFE   "LAST VALUE PRINTED"
EMPCFE .CFE   ""                    (empty string)
```

The format of the assembled bytes are:

1st byte — number of data bytes plus 1. The data bytes are those between the quotes.

next bytes — the data bytes.

last byte — always $9B. This is a checkbyte.

The examples above are equivalent to:

```
TITLE  .BYTE  19,"LAST VALUE PRINTED",$9B
EMPCFE .BYTE  1,$9B
```

The CFE is useful for defining a character field that is to be processed by subroutines. Given the address of the CFE, a subroutine can:

1. Check that the address points to the CFE rather than garbage. The checkbyte is used here.
2. Find the number of data bytes.
3. Find the data bytes themselves.

## The .DBYTE Statement

Following are examples showing all forms:

```
ABC    .DBYTE X
DEF    .DBYTE X,Y+1
```

This statement defines a two-byte constant in which the high order byte is placed first in the machine language translation. Such constants are not often used, as the 6502 processor requires that 2-byte addresses have their low order

byte first. The .WORD statement produces constants of that form.

## The .ENTRY Statement

Following are examples showing all forms:

```
.ENTRY SORTIP
.ENTRY AL1,AL2
.ENTRY "ALL"
```

Each listed label is made an ENTRY, meaning that it is capable of resolving virtuals in other assemblies. Each label that is made an ENTRY must also be defined within the source where it is so declared.

If the operand field consists of "ALL" (quotes included) then all labels defined in the source file are made ENTRYs.

An .ENTRY statement can appear anywhere in the source file, but it is normally placed near the top.

## The .OUTPI Statement

It has only one form:

```
.OUTPI packid
```

where "packid" is a 6-byte pack ID that indicates the pack (diskette) on which the assembler is to write the relocatable file. A pack ID is identical in syntax to a label, but there is no interaction between the two.

This is the only statement that is mandatory in all source files. It can be placed anywhere but is normally put near the top.

## The .VIRT8 Statement

Examples showing all forms:

```
.VIRT8 WALL
.VIRT8 ABC,LINK,VERS
```

The VIRT8 virtual was explained earlier in this chapter.

## The " = " Statement

This statement is sometimes called the "equate" statement. Following are examples:

```
LFEED  =     $9B
LFDP1  =     LFEED+1
```

```
LOOP    =       *
```

The label is defined to be the value of the expression in the operand field. A label in the label field is not mandatory, but without a label the statement is useless.

The expression must yield an absolute or relocatable value during pass 1 of the assembly process. Thus, a label used in the operand field cannot be a virtual and it must have been defined before the "=" statement is encountered.

## The "* =" Statement

Following are examples:

```
        *=      $462C
BINVAL  *=      *+2
OLDORG  *=      ABC
        *=      OLDORG
```

The location counter is set to the value of the expression in the operand field, which must yield an absolute or relocatable value during pass 1 of the assembly process.

If a label is used in the label field, it is defined to be the location counter value in effect when the assembler began processing the "*=" statement.

The "*=" must appear in the operation field with no blanks separating the "*" from the "=". In an exception to the general rule for separating fields, blanks need not separate the "*=" from the operand or label field. These exceptions apply to the "=" statement as well.

The setting of the location counter is sometimes called "setting an origin". More information on the "*=" statement was given earlier in this chapter.

## Instructions with no Operand Field

ald17

When an instruction has no operand field, any source material following the operation field is considered to be the comment field. Following are examples:

```
0009R A8        LAB1    TAY
0010R 60                RTS                     RETURN FROM READCS.
```

## Relative Branch Instructions

They are BCC, BCS, BEQ, BMI, BNE, BPL, BVC and BVS. In machine language, the operand is a 1-byte field that gives the "distance" to branch, which can range from -128 through 127. The distance is measured from the byte immediately after the branch instruction. For instance, a value of -2 (FE) would cause the instruction to branch to itself.

In assembler language, the operand field is an expression giving the destination of the branch. The assembler computes the proper "distance" value to go into the machine language instruction. The destination must be within range of the branch, and it must be within the same source file. Thus, a virtual cannot be used in a relative branch instruction (but note that virtuals can be used in the JMP and JSR instructions). Also, the expression result must have the same attribute (absolute or relocatable) a the current location counter. Here is how the relative branch operand field is processed by the assembler:

1. The operand field, a single expression, is evaluated.

2. The result must have the same attribute (absolute or relocatable) as the location counter.

3. The value "location counter + 2" is subtracted from the expression result. Recall that the location counter points to the start of the branch instruction.

4. The result of the subtraction must be between -128 and 127.

5. This one-byte result becomes the value of the assembled operand field. It can be seen in the compiled-code field in the assembler listing.

Following are relative branch examples:

```
0025R A8       BACKW  TAY
0026R F002            BEQ    FORW        A FORWARD BRANCH.
0028R 30FB            BMI    BACKW       A BACKWARD BRANCH.
002AR AA       FORW   TAX
```

## JMP and JSR Instructions

The machine language form of these instructions has a 2-byte operand field that is the absolute address of the destination. In assembler language, the operand field is an expression. In case of the JMP, the expression can be enclosed in parentheses to indicate an indirect JMP. Following are examples:

```
0044R 20VVVV          JSR    SORT        SORT THE ARRAY.
0047R 6CAB20          JMP    (ABC)       NOT USED OFTEN.
```

## Instructions with Several Addressing Modes

Instructions not already mentioned fall into this group. No single instruction has all modes. Consult a 6502 hardware manual for the modes allowable for a particular instruction. Possible modes are:

| ADDRESSING MODE | SYNTAX OF OPERAND FIELD | EXAMPLE |
|---|---|---|
| immediate | #exp1m | LDA #25 |

```
absolute            expr            LDA   ABC+3
zero-page           exp1a           LDA   ZPAGV+1

absolute,X          expr,X          LDA   ABC,X
zero-page,X         exp1a,X         LDA   ZPAGV,X

absolute,Y          expr,Y          LDA   ABC+1,Y
zero-page,Y         exp1a,Y         LDA   ZPAGV,Y

(ind,X)             (exp1m,X)       LDA   (ZPAGV,X)
(ind),Y             (exp1m),Y       LDA   (ZPAGV),Y

A-register          A               ASL   A    (only ASL, LSR, ROL, ROR)
```

where

expr is any valid expression, or, if both absolute and zero page forms of the instuction exist, then "expr" is an expression that, during pass 1, does not qualify as an "exp1a".

exp1a is an expression that yields a 1-byte result during pass 1.

exp1m is an expression that must ultimately yield a 1-byte result because there is no form of the instruction with a 2-byte operand. If the expression contains a virtual then the value is computed and checked by the linker.

# Chapter 4

## Using the Assembler

### Labelled Packs

ald8

The assembler and linker require labelled packs. Each time a file is read or written, the pack label is first checked. If it is missing or incorrect, you are informed of the problem and given a chance to mount another.

A pack is labelled by creating a file on it called PACKID that has one record that contains, beginning in the first byte, the desired pack ID. If shorter than 6 bytes, the pack ID can be ended with an EOL or blank padding to 6 bytes. Bytes following the sixth are ignored.

In the source file (specifically, the .OUTPI statement) the pack ID has the same syntax as that of a label. The first byte is a letter and remaining bytes (from 0 to 5 of them) are letters or digits.

### Assembler Capacity

Maximum source file size: 140 sectors (17,500 bytes)

Maximum number of different labels: 200

The limit on source file size is because the source file is read into memory in its entirety. Because source records vary in length, it is not possible to give a maximum number of records, but 140 sectors is typically more than 12 pages.

Note that the limit of 200 labels is only for a single assembly. The linker allows up to 512 in a build.

### Pass 1

After reading the source file, the assembler makes a total of three passes through it. During pass 1 it determines which labels are locally defined and which are virtuals, and it determines the value of the locally defined ones.

Since locally defined variables can be be defined in terms of the current location counter value, the location counter must remain defined throughout pass 1. That is why the operand field of the "*=" statement must always yield a known (absolute or relocatable) value during pass 1.

Since an instruction causes the location counter to be incremented by its

length, the length of each instruction must be determined during pass 1. Thus, automatic selection of zero-page instructions is done during pass 1, and those selections are remembered so that during subsequent passes, the location counter is incremented exactly as it was during pass 1.

## Pass 2A

It is a "dry run" of pass 2B whose sole purpose is to find out whether the source file has errors. See "Mid-Assembly Choices" below.

## Pass 2B

During this pass, the relocatable file and assembly listing are produced.

## Starting an Assembly

Chapter 2 explains how to start the assembler. The first thing it does is to ask for the source file name. Enter it and press Return. Next, the assembler asks for the pack ID of the pack containing the source file. Enter that and press Return.

> Note: After an assembly is finished, the assembler prepares to do another. When the file name and pack ID are again asked for, the previous values are displayed. They can be partially or wholely retyped or left unchanged.

The assembler then reads the source file and performs passes 1 and 2A.

## Mid-Assembly Choices

After passes 1 and 2A are complete, the assembler tells you whether or not there are errors and asks you what you would like to do with the assembly listing. You can have it printed or displayed on the screen.

If the source file has errors, the relocatable file is not built. If the source file is error free, you can suppress building of the relocatable file by holding the Option key down at the same time you type the letter indicating your choice for the assembler listing output device.

## Print Mode

Only data characters and the EOL are sent to the printer, meaning that just about any printer should work. Perforations are skipped by printing blank lines. Pages are assumed to be 66 lines in length. Before printing is begun, the printer should be set on top-of-form. After each assembly, the printer is left at top-of-form.

The XL and XE computers have a minor bug in their operating system code that

causes the printer to occasionally stop for approximately a minute. Do not do anything, as it will restart on its own. Whenever the assembler or linker is waiting on the printer, the bottom screen line has a message saying so.

## View Mode

Only the leftmost 40 bytes of the assembly listing are shown. This partial display includes everything but some of the commentary, and it is easier to visually scan than would be the case if longer lines were wrapped to a second line. View mode serves two purposes:

1. Errors can be seen and corrected before hardcopy is made. Note that source changes must be made using your word processor.

2. When a small change is made, the assembly listing need need not be reprinted in order to confirm the change.

View mode begins by filling the screen with the first 22 lines of the assembly listing. The screen (and the entire assembly process) is then advanced by using the keyboard. The bottom screen line summarizes the available functions, one of which is a "help" display. Each function is invoked by a single, unshifted letter. The functions are:

L - advance the display by one line.

S - advance the display by one screen (22 lines).

F - finish the assembly.

E - advance the display until an error message appears on the bottom display line. Error messages begin with 8 asterisks. If there are no more errors in the assembly, the display continues until the "outcome" message appears. The initial screen must be manually scanned for errors.

H - show the "help" display. To return from that display, press Return.

## Error Reporting

Following are examples of the two error message formats:

```
******** ERROR 14,25

******** xxxxxx: ERROR 22
```

The first form normally applies to the statement preceding it. The second form applies to the given label. Error codes are tabulated in Appendix A.

## The Assembly Listing

ald18

It shows each source statement along with its machine language translation (to the extent that is possible). The label, operation, operand, and comment fields are tabbed to fixed columns to improve readability. Recall that in the source file, these fields need only be separated by one or more blanks. Following are a few sample lines:

```
                    ; COMMENT STATEMENTS ARE PRINTED EXACTLY AS ENTERED
                    ;
4024R 8D3412  LABEL  LDA    XYABC        COMMENT FIELD.
```

The leftmost field is the "address" field. Next is the "compiled code" field, which shows up to the first three bytes of the machine language code (to the extent possible). Remaining fields are those of the source statement.

The address field is shown when it is relevant. In statements that reserve memory, it gives the address of the first byte of the reserved memory, which is also the location counter value in effect at the start of the statement. In the "=" statement, it gives the value of the expression in the operand field.

The following lines illustrate the address field for relocatable and absolute location counters:

```
0029R C99B              CMP    #$9B         IMMEDIATE OPERAND.
                    ;
002BR                   *=     $1000        SET AN ABSOLUTE ORIGIN.
                    ;
1000  A8                TAY                 NO-OPERAND INSTRUCTION.
```

The "R" in the address field indicates that the CMP instruction is at a relocatable address. The "*=" statement changes the location counter to "$1000, absolute". The address field in the "*=" statement shows the location counter before it is changed. The new value is not seen until the following statement.

The compiled-code field shows as much as possible. The following lines illustrate 2-byte operand field values:

```
0024R 8D3143            STA    ABS          ABSOLUTE OPERAND FIELD.
0027R 8D2D00            STA    RELOC        RELOCATABLE OPERAND FIELD.
002AR 20VVVV            JSR    SUBR         VIRTUAL OPERAND FIELD.
                    ;
002DR 3412      RELOC   .WORD  $1234        A RELOCATABLE LABEL.
4331            ABS     =      $4331        AN ABSOLUTE LABEL.
```

The final machine language value is given for absolute operands. Relocatable operand values are relative to the load point of the assembly. Note that in the instructions and the .WORD statement, the low order byte appears first. For virtual operands, "VVVV" is shown.

The following lines illustrate 1-byte operand values:

```
                        .VIRT8 ZPAGA
                    ;
```

```
0024R A531            LDA    ABS8         ABSOLUTE 1-BYTER.
0026R A5RR            LDA    .LO.RELOC    RELOCATABLE 1-BYTER.
0028R A5VV            LDA    ZPAGA+1      ZERO-PAGE INSTR. SELECTED.
                ;
002AR 00        RELOC .BYTE 0             A RELOCATABLE LABEL.
0031            ABS   =      $31          ABS. LABEL <255.
```

A 1-byte relocatable operand is permitted by the assembler only if the defining expression has a byte selection prefix. For such operands, "RR" is shown because the final value is computed by the linker.

# Chapter 5

## Linker Concepts

The linker combines relocatable files into an object file that can be loaded and executed by DOS. An execution of the linker is called a "build". A build proceeds in the following steps, which are further explained in chapter 6:

ald9

1. You enter options to control the amount of printout.

2. You enter the name and pack ID of the control file.

3. The control file is read and "remembered".

4. Relocatable files are read as directed by the control file.

5. The output file is created from the information in memory.

6. The load map is printed.

## The Linker Control File

A build is defined by giving the name and pack ID of each input (relocatable) file and the name and pack ID of the object file to be created. This information is not typed directly into the linker. Rather, a "control file" is created that contains the data. Following is a sample control file:

```
(* SAMPLE CONTROL FILE *)

/BINOUT ON TEST1: COPY.O              (* OBJECT FILE *)

/INPUTS ON RPAK1: XASA.R              (* MAIN PROGRAM *)
                  XASC.R  XASD.R      (* CHARACTER MANIPULATION *)

/INPUTS ON RPAK2: YASA.R  YASB.R      (* FILE HANDLERS *)
```

Physically, the control file is composed of records, each ending with an EOL. A record cannot be longer than 120 bytes. They are normally kept short enough to fit on one print line.

Logically, the control file is composed of statements that can be arranged in a "freeform" manner within the records. A statement can occupy multiple records as long as a single word does not cross over from one record to the next. Blanks are ignored except that they cannot appear within a word. A sequence beginning with "(*" and ending with "*)" is a comment. It can appear anywhere except in the middle of a word.

The INPUTS statement gives a pack ID and then one or more files that are to be read from that pack. The file names are separated by blanks. A complete DOS name must be given. Thus, the ".R" must be given for the relocatables.

There is no specific limit to the number of INPUTS statements or the number of file names given in a single INPUTS statement. Files are read in the order they are given in the control file.

The BINOUT statement is required to give the name and pack ID of the output file.

## Example 1: Illustrates all Linker Concepts

ald11

```
-------------- source file SOUR1 --------------

                        .OUTPI RPACK1
                    ;
                        *=     $4000
                    ;
                        .ENTRY S1STAR
                    ;
4000  A92C        S1STAR LDA    #44
4002  8D0540             STA    XYZ
                    ;
4005  00          XYZ    .BYTE  0


-------------- source file SOUR2 --------------

                        .OUTPI RPACK1
                    ;
                        .ENTRY S2STAR
                    ;
0000R AD0600      S2STAR LDA    XYZ
0003R 8EVVVV             STA    S1STAR+5
                    ;
0006R 00          XYZ    .BYTE  0


-------------- linker control file --------------

   /BINOUT ON OBJPAK: OBJFIL

   /INPUTS ON RPACK1: SOUR1.R  SOUR2.R


-------------- linker printout --------------

   400D = FINAL LOCATION COUNTER VALUE
```

```
S1STAR 4000        S2STAR 4006 R N        (load map)
```

Illustrated Here Are:

1. The linker has a location counter that parallels that of the assembler. The linker's location counter is always absolute. The first input file sets the linker's location counter to an absolute value ($4000) and then advances it in step with the memory-reserving statements in SOUR1.

   Such an absolute setting is required before any memory-reserving statements are encountered, because the linker's location counter is "undefined" at the start of a build.

2. The linker's location counter carries over from one input file to the next. The value after SOUR1.R is finished is $4006. That value remains in effect when processing of SOUR2.R starts. Thus, SOUR2.R is processed as though it began with the statement

   ```
   *=  $4006.
   ```

   This is relocation. It results from the carryover of the linker's location counter from one input file to the next. Evidence of the relocation of SOUR2.R is seen in the load map. The linker assigned a value of $4006 to the label S2STAR.

3. The label XYZ is a local label in both sources. The linker does not know about either of these labels.

4. The labels S1STAR and S2STAR are each declared to be an ENTRY. ENTRYs are seen in the load map.

5. The final location counter value is printed immediately before the load map. With appropriate ordering of the input files, this value can be made to show the highest+1 location used by the object program.

6. In the load map, S1STAR does not have "R" beside it because it was not relocated by the linker. S2STAR, which was relocated, has the "R".

7. In the load map, S2STAR has "N" beside it, meaning that it is not referenced by another relocatable file. The linker does not know whether or not an ENTRY is referenced from within the file defining it.

8. S1STAR is a virtual in SOUR2.R because it is used in that assembly but not defined in it. The linker "resolves" the virtual reference and inserts the value $4005 (S1STAR+5) into the "STA" operand.

   To see the "$4005", you could load the object program (OBJFIL) into memory and look at the two locations beginning at $400A. They will contain $0540. Recall that in the "absolute address" form of an instruction, the low order address byte is given first.

# Example 2: Techniques to be Aware of

ald19

```
------------- source file SOUR1 -------------

                        .OUTPI RPACK1
                    ;
                        .ENTRY "ALL"
                    ;
6000                INIT   =      $6000
6003                MOVCHR =      $6003
6006                READRC =      $6006
6009                A      =      $6009
600B                B      =      $600B
600D                C      =      $600D


------------- source file SOUR2 -------------

                        .OUTPI RPACK1
                    ;
0000R                   *=     $4000
                    ;
                        .ENTRY START,FLOCS2
                    ;
4000                START  =      *          PROGRAM START.
                    ;
4000  20VVVV            JSR    INIT       DO INITIALIZATION.
                    ;
4003  ADVVVV        LDA    A
4006  ACVVVV        LDY    A+1
4009  20VVVV        JSR    MOVCHR
                    ;
400C  20VVVV        JSR    PROCES
                    ;
400F                FLOCS2 =      *


------------- source file SOUR3 -------------

                    ; CAUSES PRORAM TO BE AUTO-RUN
                    ;
                    ; CAUSES NO NET CHANGE TO LOCATION COUNTER.
                    ;
                        .OUTPI RPACK1
                    ;
                        .ENTRY FLOCS3
                    ;
0000R               ORGSAV *=     $2E0       SAVE AND SET LOCCTR.
                    ;
02E0  VVVV              .WORD  START      PLANT START ADDRESS.
                    ;
```

```
02E2                    *=      ORGSAV      RESTORE LOCCTR.
                ;
0000R           FLOCS3 =        *
```

---------- source file SOUR4 ----------

```
                        .OUTPI RPACK1
                ;
                        .ENTRY PROCES,FLOCS4
                ;
0000R           PROCES =        *           ENTRY POINT.
                ;
0000R  ADVVVV           LDA     C
0003R  297F             AND     #$7F
0005R  20VVVV           JSR     READRC
                ;
0008R           FLOCS4 =        *
```

---------- linker control file ----------

```
/BINOUT ON OBJPAK: OBJFIL

/INPUTS ON RPACK1: SOUR1.R (* EQUATES *)
                   SOUR2.R (* MAIN PROGRAM *)
                   SOUR3.R (* AUTO-RUN CAUSER *)
                   SOUR4.R (* PROCES *)
```

---------- linker printout ----------

```
4017 = FINAL LOCATION COUNTER VALUE

A      6009          B      600B  N      C      600D
FLOCS2 400F  N       FLOCS3 400F RN       FLOCS4 4017 RN
INIT   6000          MOVCHR 6003          PROCES 400F R
READRC 6006          START  4000
```

Note the Following:

1. SOUR1 contains nothing but "equates". It does not use the location counter at all. Thus, the location counter remains undefined after processing of SOUR1 is complete.

2. The FLOCS2, FLOCS3 and FLOCS4 labels are included to show the location counter value at the end of each file. Note that SOUR1 cannot have a "FLOCS1 = *" since the location counter is not defined within SOUR1.

3. SOUR3 causes the output file to be "auto-run", meaning that after it is loaded by DOS, control is automatically transferred to "START". See SSOUQ in the sample program for more information on this feature of

Atari's DOS.

SOUR3 also has statements that save and restore the location counter. Without those statements, SOUR3 could not be placed before SOUR4, as it would cause SOUR4 to be loaded beginning at $2E2 rather than following SOUR2 as was intended.

4. The control file contains informative commentary.

# Chapter 6

## Using the Linker

## Linker Capacity

ald12

1. Maximum number of different ENTRY and virtual names in a single assembly is 256.

2. Maximum number of different ENTRY and virtual names in an entire set of linker inputs is 512.

3. Maximum number of input files is in excess of 100.

4. Maximum object program size is approximately 8,000 bytes of instructions and constants, as explained next.

## Instructions and Constants

Instructions and constants represent memory that is reserved and initialized when the program is loaded into memory. Statements that do this are the 6502 instructions, .WORD, .BYTE and .CFE.

Memory that is reserved but not initialized does not fall in the category "instructions and constants". Such reservations are done using the "*=" statement in the following manner:

```
ARRAY  *=   *+100     100-BYTE ARRAY.
```

Consider the following group of instructions:

```
ABC    LDA    DATA
       STA    ARRAY
DATA   .BYTE  22
ARRAY  *=     *+1000
```

It has 7 bytes of instructions and constants and reserves a total of 1007 bytes of memory. The 1000-byte reservation does not use up linker capacity.

## The Internal Workarea

Most of each relocatable file is held in the linker's "internal workarea". The workarea size is fixed at approximately 14,000 bytes. For every byte of instructions and constants, approximately 1.7 bytes of workarea space is required. The limit given above of 8,000 bytes of instructions and data was

computed by dividing 14,000 by 1.7.

At the end of each build, the internal workarea's capacity and the number of bytes used by the build are displayed. Thus, you can gauge how close a build is coming to the linker's capacity.

## Starting a Build

ald10

Chapter 2 explains how to start execution of the linker. The linker begins with the following display:

```
                LINKER

          (C) Copyright 1985 by
           Six Forks Software
           All Rights Reserved

V - DON'T USE PRINTER.  USE SCREEN.
C - DON'T SHOW ALL OF CONTROL FILE.
N - DON'T SHOW NUMERICALLY SORTED LOAD MAP.
U - DON'T SHOW UNREFERENCED NAMES.

PRESS 'OPTION' TO SUPPRESS OUTPUT FILE

TYPE 1 OR MORE (OR NONE)

(cursor appears on this line)
```

Type the desired letters and press Return. If the OPTION key is held down when Return is pressed, the output file is not written. That is a convenience if, for instance, the build is being re-run to get more of the printout. If the output file is suppressed, a line saying so is printed following the load maps.

When finished with a build, the linker prepares to do another. As with the assembler, previously typed input is displayed and can be changed or left unchanged as desired. Output file suppression does not carry over from one build to the next.

Here is what the option letters mean:

V - if present, output is directed to the screen in exactly the same manner as in the assembler. Refer to the "Print Mode" and "View Mode" sections in chapter 4.

C - If present, only the first 5 lines of the control file are printed, followed by a message saying "REMAININDER OF CONTROL FILE NOT PRINTED". This option and other printout reduction options are useful because as a program is being developed, the same (or almost the same) build is re-run many times.

N - if present, the numerically sorted load map is NOT printed. The

alphabetically sorted one is always printed unless errors are detected.

U - if present, unreferenced ENTRYs are omitted from the load map.

## Control File Reading

The control file is completely read, checked, and remembered before relocatable file reading begins. If errors are found then the build is terminated after control file reading is done.

## Error Handling

Errors are checked for during all phases. If an error is detected before output file building begins, the output file is not built. Also, if any errors are detected, the load map is not printed since it would not be of assistence in correcting the errors.

Error codes are tabulated in Appendix A. When an error pertains to a particular relocatable file, the file name is included in the message. When an error pertains to a particular label, the label is included in the message.

## Input File Reading

Input files are read in the order they are given in the control file. You are prompted to mount the necessary packs as they are needed.

## Output File Building

All external labels must be defined when input file reading is complete. If not, undefined ones are listed and processing terminated.

One-byte field overflow errors are detected during this phase. They are errors in which the result of an expression containing a virtual is greater than 255, and the result is being used in a one-byte field.

## The Load Map

It is a list of all external labels and the information associate with each. The following information is shown for each label:

a. The 6-byte name.

b. The 16-bit absolute value it denotes, shown in hex.

c. If the value was relocated by the linker, "R" is printed.

d. If the value was not referenced by a relocatable file other than the one in which it was defined then "N" is printed.

NOTE: unreferenced labels occur for various reasons, especially in multi-group programs. They are normally not errors and should not be eliminated just because they are unreferenced.

The load map is required for debugging, as it shows where the linker placed the relocatable code in memory. Actually, the load map shows this clearly only if the beginning of each assembly has an ENTRY label on it, but that is normally the case because a memory-reserving assembly is likely to be either:

a. A subroutine (or several subroutines). The entry point of the first subroutine is normally at the start of the assembly.

b. Global data. All of these data items are likely to be ENTRYs.

The load map is printed twice: alphabetically sorted and, unless suppressed with the "N" option, numerically sorted. The numerically sorted map is useful because it gives a quick means of determining the ENTRY labels that a given absolute address is near. That is sometimes useful during debugging.

# Chapter 7

## Multi-Group Programs

It is sometimes useful or necessary to divide a program into multiple builds. In such a program, each build is referred to as a GROUP. The complete program consists of all of the groups loaded into memory at once. Note that to DOS and the linker, a group is not different than any other object file.

ald13

Division of a program into multiple groups is necessary if its si e (the amount of instructions and constants, as explained in Chapter 6) exceeds the linker's capacity. Whether or not necessary, it is often beneficial for the following reasons:

1. When a change is made, only the affected groups need be rebuilt. As a program grows, a growing number of assemblies become fully debugged and stable. Those can be collected into separate groups and thus not have to be run through the linker time and time again.

2. A group can be used in more than one program. You are likely to create a group that contains routines useful to practically every program you develop. Such a group could be thought of as a "foundation". Functionally speaking, Atari's operating system in ROM and the DOS routines in low memory are foundation groups.

When developing a multi-group program, you have the following responsibilities:

1. Memory assignment for each group.

2. Communication between the groups.

3. Loading each group into memory prior to execution of the program.

## Memory Assignment for each Group

As with any object file, the memory usage of a group is controlled by the source code. To simplify your responsibilities, a group should have the following structure when possible:

1. The first assembly in the linker control file begins with an absolute "*=" statement. That statement defines the starting address of the group.

2. Remaining assemblies in the group contain only relocatable code. Thus, the linker places them in ascending memory locations following the first assembly.

It is useful to define an ENTRY label at the end of the last assembly in the group. In the load map, that label shows the highest location (actually the "highest+1" location) used by the group. Some might prefer to put this label definition in an assembly by itself.

It might also be noted that, with appropriate ordering of the linker input files, the final location counter value printed by the linker can be made to show the "highest+1" address.

You must manually arrange the groups in memory so that they do not fall on top of each other, but in practice that is not difficult, especially if the suggestions in this chapter are followed. Here are some reasons that it is not difficult:

1. There are seldom more than 3 or 4 groups, and some of them will seldom require modification.

2. Because of the small number of groups, it is not difficult to manually insure that the groups don't overlap. It is not unreasonable to leave several hundred bytes between each group to allow for future expansion. Given that amount of space following a "stable" group, minor changes should seldom require groups to be moved.

3. Moving a group is easy. All that is necessary is to change the initial "*=" statement, reassemble that source, and rebuild the group. Also, other groups referencing that group must be rebuilt.

4. Here is an easy way to initially arrange groups in memory:

   a. Initially place each group so it begins at location 0. Build the groups like this. The assemblies and builds can be done using view mode, thus avoiding hardcopy.

   b. The length of each group is simply its "highest+1" address.

   c. Based on the lengths and desired separations, decide where each group should start. Set the "*=" statements, reassemble those sources, and rebuild each group.

5. To establish communication between the groups, use the techniques described in the remainder of this chapter.

## The Communication Vector

A group typically contains certain subroutines and data areas that must be acccessed by other groups. Collect those entry points and data areas into the first assembly in the group, just after the "*=" statement defining the group's origin. Make that assembly contain nothing but these items. This assembly is called the "communication vector".

Other groups reference those entry points and data areas by simply using their

names as virtuals. The virtuals are resolved by a relocatable file called an "entry definitions file" that contains nothing but "=" statements defining the names. The use of a "communication vector" and an "entry definitions file" has the following benefits:

1. Changes can be made to the non-first assemblies in the group without altering the addresses of the externally-referenced entry points and data areas. Thus, bugs can be fixed or changes made to the "body" of the group with requiring other groups to be rebuilt.

2. The entry definitions file can be automatically produced when the communications vector is assembled. That is done using the .EOUT statement, explained below.

3. The "transparent jump" feature in the assembler prevents a problem that would otherwise accompany the placement of subroutine entry points in the communication vector. This feature is also explained below.

Here is an example of a communication vector that does not use the .EOUT or transparent jump features:

```
                        .OUTPI RPACK1
                ;
                        .ENTRY "ALL"
                ;
0000R                   *=     $4430
                ;
4430  4CVVVV    FNGET   JMP    FNG      FNGET ENTRY POINT.
4433  4CVVVV    DOPAS1  JMP    DOP1     DOPAS1 ENTRY POINT.
                ;
4436  00        NBYTES  .BYTE  00       GLOBAL DATA AREA.
4437  0000      ENDADR  .WORD  00       GLOBAL DATA AREA.
```

This group has two subroutines and two data areas that are to be visible to other groups. In the linker control file that builds the group, the communication vector is given first. Remaining assemblies contain the body of the group. They define the labels FNG and DOP1.

Note that FNGET, DOPAS1, NBYTES and ENDADR are visible to other assemblies within the group because they are ENTRYs in the ".R" file. To make these labels visible to other groups, an entry definitions file such as the following is created:

```
                        .OUTPI RPACK1
                ;
                        .ENTRY "ALL"
                ;
4430            FNGET   =      $4430
4433            DOPAS1  =      $4433
4436            NBYTES  =      $4436
4437            ENDADR  =      $4437
```

This assembly is used in the builds of the other groups.

## The .EOUT Statement

This statement causes the entry definitions file (relocatable version only) to be created automatically when the communications vector source is assembled. Thus, when the .EOUT statement is present, two relocatable files are produced. Here are characteristics of the .EOUT statement:

1. It can appear anywhere in the source file, but is normally placed near the top.

2. It does not affect the contents of the ".R" file.

3. The entry definitions file is created on the same pack as the ".R" file. The name of the entry definitions file is formed by adding ".E" to the source file name.

4. The entry definitions file is a relocatable file. No corresponding source file is created.

5. The entry definitions file has the following contents (equivalent source code shown):

```
        .OUTPI rpack        same pack ID as ".R" file.
;
        .ENTRY "ALL"
;
label1 =      $val1
label2 =      $val2
.
.
.
labeln =      $valn
```

   For each ENTRY label, the equivalent of a "=" statement is generated that defines the label so that it denotes the same value as in the ".R" file. Note that the ".E" file contains only ENTRY definitions. It reserves no memory and does not use the location counter.

## The Transparent Jump Statement

In the above example, the subroutines callable through the communication vector have two names. To fully comprehend this nuisance, consider the likely history of subroutine FNGET. It is created without vectored calling in mind. Thus, the initial version of the source file contains FNGET as an ENTRY. As the amount of source code grows, it becomes beneficial to put FNGET into a group and put its entry point into the communications vector.

To do that, the name FNGET must be defined in the communications vector, meaning that another name (FNG in our example) must be used in the subroutine's source file. That in turn necessitates changing the FNGET source, which in

turn means that it will no longer work in earlier builds since it now has the "wrong" name.

This problem is eliminated with the transparent jump feature. Here is our communications vector written to use transparent jumping:

```
                        .OUTPI RPACK1
                    ;
                        .ENTRY "ALL"
                    ;
                        .EOUT           CREATE THE ENT. DEFS. FILE.
                    ;
0000R                   *=      $4430   STARTING POINT FOR GROUP.
                    ;
4430 4CVVVV         =       JMP     FNGET   A TRANSPARENT JUMP.
4433 4CVVVV         =       JMP     DOPAS1  A TRANSPARENT JUMP.
                    ;
4436 00             NBYTES .BYTE    00      GLOBAL DATA AREA.
4437 0000           ENDADR .WORD    00      GLOBAL DATA AREA.
                    ;
```

An "=" has been added to the label field of the JMP instructions. The .EOUT has been added.

A transparent jump is indicated by putting "=" into the label of a JMP instruction. Transparent jumping has the following characteristics:

1. It can be used only in a file containing a .EOUT statement. That is because its only effect is in the entry definitions file. It has no effect on the ".R" file.

2. The operand field of a transparent JMP instruction can consist of only a single label that must be a virtual.

3. the result of the transparent jump is:

   a. An ENTRY label definition with the following properties is placed in the ".E" file :

      label: label in the operand field

      the label denotes: address of the JMP instruction

   b. The JMP instruction, ignoring the "=" in the label field, is assembled in the ".R" file in the normal manner.

Here is how transparent jumping affects subroutine FNGET in our example:

1. The source file containing subroutine FNGET can use the name FNGET for its entry point.

2. The address in the operand field of the "= JMP FNGET" instruction points to the entry point within the source file containing subroutine

FNGET.

3. In the "R." file produced by the assembly of the communications vector source, FNGET is a virtual.

4. In the "E." file produced by that assembly, FNGET is an ENTRY that denotes the "= JMP FNGET" instruction.

5. The net result is that callers to FNGET, both inside and outside of the group where it is defined, can use the name FNGET. From inside the group, control is transferred directly to the subroutine. From outside the group, control goes through the communications vector.

## Executing a Multi-Group Program

During debugging it is normally easiest to load the groups individually and then manually branch to the program's starting point.

When development is complete, the program can be packaged into a single object file by concatenating all of the groups into a single file. Atari discusses this process under the topics "appending" and "compound files".

Be aware of the flaw in Atari's Assembler/Editor cartridge which causes it to not accept files concatenated by the "append" option of Copy. However, those files are accepted by the Load command.

The problem is that the cartridge expects "pieces" after the first to not have the initial pattern of $FF,$FF. The "append" option of the Save command produces acceptable files. It is easy to write a Basic program to do the combining. Simply concatenate the files, excluding the first two bytes of those after the first.

# Chapter 8

# The Sample Program

## Files on the Distribution Diskette

ald16

Only the source files and linker control file are supplied. They are:

| | |
|---|---|
| LNKSP1 | linker control file to build the object program. |
| SSOUA | Main program |
| SSOUB | SUBTR, COLDIS. Trigger input, set color. |
| SSOUC | SUBFW, SUBBK, BLUDIS. Forward, backward. Set background color. |
| SSOUD | SUBLF, SUBRT, FLUDIS. Left, right. Set foreground color. |
| SSOUE | DISHB. Display byte in hex. |
| SSOUF | CFEFSF. Move CFE to screen. |
| SSOUG | Global areas |
| SSOUH | Page 0 variables, system equates |
| SSOUI | Screen and display list |
| SSOUJ | Defines origin for program. |
| SSOUK | TERROR, RTSJMP. Terminal errors, All RTS. |
| SSOUL | DISCRN. Set up display list and screen. |
| SSOUM | GETAR, AWSGAR. Get subroutine arguments. |
| SSOUN | AWSIS, AWRFS. Save and restore AL, AWD1-4. |
| SSOUO | ATAVID, VIDATA. Convert atascii to/from video. |
| SSOUP | CV1B2H. Convert byte to two atascii hex digits. |
| SSOUQ | Makes program auto-run |

## Assembling and Linking the Sample Program

It takes approximately 14 minutes to do the assemblies (assuming no hardcopy is made) and less than a minute to do the build.

step 1: Make a scratch pack for receiving the outputs of the assemblies and build. Format it and put on DOS and DUP. Make it have a pack ID of SCRAT. That is done by creating a file called PACKID that has one record whose contents are "SCRAT" (quotes excluded) followed by an EOL.

step 2: Assemble each of the 17 source files. To assemble SSOUA, proceed as follows:

a. Start the assembler as explained in Chapter 2.

b. Insert the distribution diskette. Its Pack ID is "ALPAK1". If you fail to do this now, you will be instructed to do so when the pack is actually needed.

c. Type SSOUA followed by Return.

d. Type ALPAK1 followed by Return.

e. When the next typein is requested, first insert your SCRAT pack.

f. Type the letter B. This causes the assembly listing to be displayed on the screen rather than printed. Chapter 4 explains how to print the assembly listings, but be aware that they occupy approximately 50 pages.

g. Shortly, the bottom screen line will display "TYPE DISPLAY CONTROL OR H (HELP)". Type F to cause the assembly process to complete without further pauses for viewing the assembly listing.

f. This assembly is now done. Press Return as instructed. The assembler then prepares to do another. Repeat steps b through this step until you have done all 17 sources. When the pack ID is asked for, and the value "ALPAK1" is showing, just press Return to use that value again.

The SCRAT pack now has all 17 relocatables, named SSOUA.R through SSOUQ.R.

step 3: Use the linker to build the object program. Proceed as follows:

a. Start the linker.

b. Mount the distribution diskette (the ALPAK1 pack).

c. Type V followed by Return. This causes the printout to be directed to the screen instead of the printer.

d. Type LNKSP1 followed by Return.

e. Type ALPAK1 followed by Return.

f. Shortly, the bottom screen line will display "TYPE DISPLAY CONTROL OR H (HELP)" as it did during the assemblies. Type F to allow the build to complete without further pauses for viewing.

g. Mount the SCRAT pack as instructed. Choose the "A" option of the pack mount display.

The object program, called SAMP1.O, is now built on the SCRAT pack.

## Executing the Sample Program

Reboot from the SCRAT pack. On XL and XE models, hold down the Option key. Start the program by typing:

L followed by Return
SAMP1.O followed by Return

That is, do a normal DOS load. Since the program is auto-run, it starts automatically.

The program creates a mode 0 screen display of all 256 character values and allows you to vary the color and intensities through all possible values. The varying is done with the joystick as follows:

| | |
|---|---|
| forward/backward | increments/decrements the background intensity. The current intensity value is shown on the screen. |
| right/left | increments/decrements the character (foreground) intensity. The current intensity value is shown on the screen. |
| trigger | Circularly advances the color. The numeric color code along with the color name is shown on the screen. |

# Chapter 9

## Programming Suggestions

1. Structure your program as explained in the "Big Picture" section of chapter 1. Don't hesitate to package a portion of logic into a subroutine even if it is called only once.

ald26

2. Use commentary as illustrated by the sample program. Use the comment statement frequently, and space out the listing with "blank" comments. Don't try to say everything in the comment field in each statement, as there's not enough room.

3. Write assembler language instructions in small groups that are comparable to high level language statements. Set these groups off with commentary.

4. Do not X or Y for long term storage, as they are too often needed for short-term use. Don't hesitate to store a register in memory and bring it back when needed. Don't worry about using an extra instruction or two. Except in loops that are critical to performance, never worry about an instruction or two.

5. At the front of each subroutine, describe its calling sequence. Subroutine entry points should be clearly indicated.

6. The BIT instruction and the (ind,X) addressing mode are not often useful.

7. Think about ways of making a program easier to debug and understand. Make that a design objective.

8. Don't use assembler language in complex ways unless there's a good reason. Don't use the stack for much more than JSR/RTS operation and saving X and Y. A subroutine should save X and Y unless those registers are used to pass an output value.

9. Become an expert at compares of one and two-byte numbers.

10. Put internal checks into the program and have a "terminal error" routine to call when one fails. That routine should "halt" the program in such a waythat you can a) tell that it was called, b) tell where it was called from, and c) start up your debugger.

11. Have all subroutine returns branch to a common label (a virtual) that does the RTS (as is done in the sample program). That RTS routine becomes a place that control passes through frequently. It can be a useful place to put debug code.

12. Save critical values in local areas if it would assist in debugging.

13. Use no absolute addresses in instruction operands. Define ENTRY labels for those addresses and put the definitions in separate sources. Don't forget to use the VIRT8 declaration for zero-page labels.

12. Make most of your assemblies contain nothing but relocatable code. Put absolute code and hardware-related definitions into separate assemblies.

14. Keep source files on one set of packs and relocatables on another. This has two benefits:

    a. Many relocatables can be put onto a single pack, thus minimizing the number of pack mounts required when the linker is run.

    b. The source packs aren't written on each time an assembly is done.

15. As you accumulate source files, look for opportunities to form groups as explained in chapter 7.

# Appendix A

## Error Codes

This list is for both assembler and linker. Some codes can occur in both.

ald14

10 Identifier (label or pack ID) longer than 6 bytes.
11 Missing close quote on string.
12 Excessively long string. Maximum is 100 bytes.
15 Bad number. No digits after "$".
16 Hex number has more than 4 digits after the "$".
17 Decimal number too large. Maximum value is 65535.
19 Bad label field. It must be a label or "=". Possibly, you have started the operation field in the first record byte.
20 Label is longer than 6 bytes.
22 Label defined more than once.
23 Value of label is not computable.
24 "*" in operation field must be first byte of "*=".
25 Operand field of "*=" statement must be an absolute or relocatable value.
26 Operand field of "*=" statement cannot have a byte selection prefix.
27 Operand field of of "*=" statement must be an absolute or relocatable value during pass 1 of the assembly process.
28 Syntax error in expression in operand field.
29 Label too long in an expression.
30 Mis-formed byte selection prefix. I+ can be .LO. or .HI.
31 Same as #30.
32 Non-allowable addition in an expression. See chapter 3.
33 Non-allowable subtraction in an expression. See chapter 3.
34 Non-allowable multiplication in an expression. See chapter 3.
35 Non-allowable division in an expression. See chapter 3.
36 Attempt to divide by 0.
37 Expression result is uncomputable.
38 Only an absolute or relocatable value can be assigned to a label.
39 Ending "." in byte selection prefix is missing.
40 Unrecognizable operand field
41 In the instructions of the form "LDA (expr,X)" or "LDA (expr),Y", the expression is not followed by a comma or close parenthesis respectively.
42 In the instructions mentioned in #41, only "X" or "Y" can follow the comma.
43 In instructions of the form "LDA (expr,?), the "?" must be "X".
44 In instructions of the form "LDA (expr),?", the "?" must be "Y".
46 This statement has a mandatory one-byte operand field, and the expression result is an absolute value that is greater than 255. That is, it will not fit into one byte.
47 This statement has a mandatory one-byte operand field, and the expression is relocatable with no byte selection prefix. Relocatable values can be used in one-byte operand fields only if byte selection is specified.
49 Operation field is not recognizable.

50 Operation field is undefined.

51 A character constant in an expression can be only one byte long. For example, "LDA #'A'" is valid.

52 The form of the instruction you have written is not valid for that instruction. For example, "DEC ABC,Y" is not permitted because the DEC instruction does not allow indexing by Y.

53 In this indirect form of the JMP instruction, the close parenthesis is missing.

55 Problem writing output file. It was successfully opened. The problem occurred after that. Could be that the drive went off-line.

56 Operand field has unrecognizable material following it. That material is shown in the comment field in the assembly listing.

57 Too many different labels in this assembly. Assembler capacity is 200.

58 The operand field of this branch instruction has an unknown value.

59 The operand field of a branch instruction cannot contain a virtual.

60 Operand field cannot be computed because location counter is undefined.

61 Branch instruction has invalid operand field. Location counter is relocatable, but the result of the expression is absolute

62 Branch instruction has invalid operand field. Location counter is absolute, but the result of the expression is relocatable.

63 Branch destination is out of range. The value "expr-locctr-2" must be between -128 and 127.

64 The given label has a value assigned to it, but the value is unknown.

65 The given label is declared to be an ENTRY, but it has no value assigned to it. That is, it has not appeared in a label field.

66 The given label is declared to be an ENTRY, and it has a value assigned to it, but the value is unknown.

67 The given label appears in a VIRT8 statement, but it has a value assigned, meaning it is not a virtual.

70 Error in .OUTPI statement. The operand field must be the output pack ID, which must be a name that has the same syntax as a label.

71 Error in .OUTPI statement. Pack ID is longer than 6 bytes.

72 Error in .VIRT8 statement. The operand field must be a list of labels separated by commas.

73 Error in .VIRT8 statement. Labels must be no longer than 6 bytes.

74 Assembly contains more than one '.ENTRY "ALL"' statement.

76 Syntax error in the '.ENTRY "ALL"' statement.

77 The '.ENTRY "ALL"' statement cannot appear along with any other .ENTRY statements.

78 Multiply defined transparent jump label. For example, in the sequence

```
=    JMP    ABC
=    JMP    ABC
```

the 2nd JMP will generate the error because in the ".E" file, ABC would be multiply defined. See Chapter 7.

79 The location counter is undefined for this transparent jump command.

80 Invalid operand field in transparent JMP (a JMP that has "=" in the label field). The operand field must be a single label that is a virtual.

81 Error in .CFE statement. Its operand field must be a single character string from zero to 100 bytes in length.

82 Assembly has more than one .EOUT statement.

83 Assembly contains too many labels. Maximum number is 200.

84 A transparent jump is not permitted unless an .EOUT statement is present. A transparent jump is a JMP instruction that has "=" in the label field.

See chapter 7.

85 A "=" is permitted in the label field of only the JMP instuction.

86 Operand field of the "=" statement cannot have a byte selection prefix unless the expression yields an absolute value.

100 Problem reading control file from disk. It was successfully opened. The problem occurred in the read itself. ald15

101 Missing "/" at start of statement.

102 Statement name does not follow "/".

103 Undefined statement name after the "/".

104 Improperly formed file name. It must begin with a name (an identifier).

105 Improperly formed file name. The first part must not be longer than 8 bytes.

106 Improperly formed file name. There is a period after the first part, but no extension following that.

107 Improperly formed file name. The extension must not be longer than 3 bytes.

108 Improperly formed file name.

109 "ON" expected but not found.

110 A pack ID must follow "ON".

111 Pack ID is longer than 6 bytes.

112 Missing ":" after pack ID, or other problem in "ON pack ID" construction.

113 Bad file name.

114 More than one BINOUT statement is present.

115 Bad "ON packid" portion of BINOUT statement.

116 Bad file name in BINOUT statement.

120 Control file does not contain a BINOUT statement.

121 Control file contains no INPUTS statements.

122 Cannot open relocatable file, probably because it could not be found. Remember that in the linker control file, the ".R" must be included in relocatable file names.

123 Error reading relocatable file. It was opened successfully. The problem occurred in the read itself.

124 Attempt to read input file beyond EOF. Likely cause: error within the relocatable file.

125 Missing S-O-A command. See note 1. Probably means that the input file is not a relocatable file.

126 The given label is multiply defined.

127 Local ordinal value out of range. See note 1.

128 Error in one-byte constant (.BYTE) containing a virtual reference. The final value is greater than 255, and thus will not fit into one byte.

129 Error in instruction with one-byte operand field that contains a virtual reference. The final value is greater than 255, and thus will not fit into one byte.

131 Too many different labels. The linker capacity is 512.

132 Same as #131.

133 Same as #131.

134 Hit EOF at a bad time while processing relocatable file. See note 1.

135 Could not open output file.

136 Internal workarea is full. See chapter 6.

137 Bad ROP. See note 1.

138 Relocatable location counter setting (the "*=" statement) was encountered before an absolute location counter setting was encountered.

139 An absolute location counter setting (the "*=" statement) must precede

memory-reserving statements.

140 An ENTRY label having a relocatable value occurred before an absolute location counter setting. Similiar to #139.

141 Hit EOF in relocatable file at a bad time. See note 1.

142 Same as #141.

143 Bad ROP. See note 1.

144 Same as #143.

146 Bad local ordinal value. See note 1.

147 This label is an undefined virtual.

148 Local ordinal value is out of range. See note 1.

149 Same as #141.

150 One-byte field (either in .BYTE or instruction with one-byte operand field) is a relocatable value, and the final value is greater than 255. This error cannot occur with the present assembler because it does not allow a one-byte field to be relocatable unless it contains a byte selection prefix.

Note 1: The relocatable file has an error in it, which can be due to:

a. The file is not a relocatable file.

b. The file was damaged after it was created by the assembler.

c. The assembler malfunctioned.

d. The relocatable file came from a language processor other than the assembler, and that malfunctioned.

# Appendix B

## Specifications

1. Hardware requirements: 48k, disk, printer ald27

2. Other requirements:

   a. Word processor for creating and editing source files.

   b. General knowledge of assembler language programming of the Atari

   c. Debugger. We recommend the one in Atari's Assembler/Editor cartridge.

3. The delivered product includes: Assembler, Linker, Sample program and Reference manual. The software is supplied on disk.

4. Relocatable code and external labels are supported. Expressions containing a virtual must reduce to "virtual plus or minus a constant" and cannot be used in the "*=" or "=" statements.

5. Labeled packs are required, which insures that the correct packs ar mounted during the assembly and linking processes. A pack is labelled by creating a file on it called PACKID that contains the desired pack ID.

6. Source statements are modelled after Atari's Assembler/Editor. Statements do not have line numbers. The maximum source program size is approximately 140 sectors (17,500 bytes). The maximum number of different labels that can appear in a source file is 200.

7. The assembly listing can be directed to either the printer or the screen. Before the use makes this choice, the assembler indicates whether or not there are errors. Thus, source errors can be corrected before hardcopy is made.

8. Parameters for the linker (file names and pack IDs) are given in a "control file". When the linker is run, only the control file name and pack ID need be typed.

9. The linker can handle over 100 input files, up to 512 external labels, and up to 8,000 bytes of instructions and constants. Features are provided for creating larger programs by using multiple executions of the linker.

10. The linker produces an alphabetically sorted load map and, optionally, a numerically sorted one.

11. Linker printout can also be directed to the screen.

12. Performance data:

    During assembly, source files are read at maximum disk speed. The remainder of the assembly is unlikely to take more than 15 seconds (excluding time to print hardcopy, which is optional).

    Linker performance can be illustrated by citing the sample program that we supply. It takes the linker 25 seconds to read the control file and 17 relocatable files and write the object file. The sample program contains approximately 500 instructions, 48 external labels, and 140 references to those labels. The source files, which contain a large amount of commentary, require 50 pages to list.

    The assembly and linking process is fast enough to eliminate the need for machine language patch files.

13. The assembler and linker have been tested on:

    a. 800 and 130XE computers.

    b. 810 and 1050 disk drives.

    c. DOS release 2.0. Only standard entry points are used, so the programs should run on newer releases as well.

    d. Okidata 82A printer. Only data characters and the EOL are sent to the printer, so no printer dependence should exist.