

A.P.P.L.E.

A
P
P
L
E

P
U
G
E
T
S
O
U
N
D

P
R
O
G
R
A
M

L
I
B
R
A
R
Y

E
X
C
H
A
N
G
E

PRESENTS

**INTEGER BASIC +
PLUS
TED][+**

INTEGER BASIC + *plus* TED II +

DISCLAIMER

This manual and the accompanying diskette are available only to members of Apple Pugetsound Program Library Exchange. Every effort has been made to provide error free programs and documentation. Inevitably some may remain. A.P.P.L.E. denies any responsibility for loss or damage of programs or equipment, direct or indirect, even if A.P.P.L.E. has been advised of the possibility of such damages.

This manual is copyrighted with all rights reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Apple Pugetsound Program Library Exchange.

Entire contents Copyright (c) 1980 by Apple Pugetsound Program Library Exchange, a Washington State non-profit Corporation. All rights reserved.

Integer Basic Copyright (c) 1976 by Apple Computer, Inc. Integer Basic Plus produced under license agreement with Apple Computer, Inc. and Copyright (c) 1980 by Apple Pugetsound Program Library Exchange.

DOS 3.2.1 Copyright (c) 1979 by Apple Computer, Inc. DOS 3.2.1 Plus Produced under license agreement with Apple Computer, Inc. and Copyright (c) 1980 by Apple Pugetsound Program Library Exchange.

Mini-Assembler, Floating Point Routines, Multiply/Divide Routines, Sweet 16 and PRDEC routine Copyright (c) 1977 by Apple Computer and distributed under license agreement by Apple Pugetsound Program Library Exchange.

TED II + and Scruncher Copyright (c) 1980 by Ken Smith.

INTEGER BASIC + *plus* TED II +

TABLE OF CONTENTS

Introduction	2
Programs on this diskette	2
Part I The Integer Basic Plus System	3
Overview	3
Getting Started with Integer Basic Plus	4
Transferring Integer to other Diskettes	5
USER (X) Functions	5
Table of USER (X) Functions	6
DOS ONERR GOTO Function	7
Defining your own Functions	8
Stop List Feature	9
Program Compatibility	9
Part II TED II + Editor/Assembler	10
Assembly Language Whys and Wherefors	10
Additional Reading	11
Background and Features	12
Executive Mode Commands	13
The Edit Module	15
Delimiters	16
Abbreviations	16
Command Syntax	16
Edit Module Commands	17
System Commands	17
Entry Commands	18
Edit Mode Sub-Commands	20
Pseudo-ops	21
Summary of TED Pseudo-ops	21
Group A – Optional 6502 Instructions	21
Group B – TED System Pseudo-ops	21
Group C – Functional Pseudo-ops	23
Addressing Modes and Arithmetic Operators	24
Removing Excess Spaces from the Source File	25
TED Pointers and Memory Map	26
Part III Subsidiary and Utility Programs	27
Programs Relating to Integer Basic	27
Autostart ROM Supplement	27
TED Related Programs	28
Scruncher	28
Universal Disassembler	28
Superdis!	28
Features	29
Method of Use	29
General Comments	30
Compatibility with TED II +	31
How to Handle a Large Program	32
Integer Basic F4 ROM Utilities	
	Inside Back Cover

INTRODUCTION

Integer Basic + Plus TED II + is a dual purpose diskette. TED II + requires Integer Basic IN ROM or on the MAIN BOARD. A 48K Apple II + does not have sufficient memory available to support RAM Integer Basic, TED and a TED source file.

The utility programs are the reason that both TED and Integer have been included on the same diskette. The utilities have been furnished in both object (runable) code and TED source format. Those Apple II + owners who do not have Integer Basic in ROM may use TED on a friends or dealers machine to relocate the object code for the utility programs if required.

The inclusion of the utilities in source format makes it possible for the first time to easily modify and relocate these programs to suit your own needs. The availability of the object code to Apple II + owners without Integer ROM makes it possible for them to CALL these utilities as needed from an Integer Basic or assembly language program, thus upgrading their Apple II + to an Apple II.

This manual is divided into three sections: Integer Basic, The Utilities and TED II +. Programs on this diskette are as follows:

A HELLO
A INTEGER
B ROM INTEGER
B UPDATE 3.2.1 +
I CONVERT
T USER DEFINE
I DEMOS

I CHR\$ FUNCTION
I STR\$ FUNCTION
I VAL FUNCTION
I STRING ARRAY FUNCTION
I INTEGER BASIC POST-EDITOR

I TED II +
B SCRUNCHER
B UNIVERSAL DISASSEMBLER
B SUPERDIS!

A SUPPLEMENT. DOC
B SUPPLEMENT

B MINI-ASSEMBLER.S
B MINI-ASSEMBLER.O
B FLOATING POINT RTNS.S
B FLOATING POINT RTNS.O
B MULTIPLY/DIVIDE RTNS.S
B MULTIPLY/DIVIDE RTNS.O
B MULTIPLY DEMO.S
B MULTIPLY DEMO.O
B M/D RTNS W/DEMO.S
B M/D RTNS W/DEMO.O
B SWEET 16.S
B SWEET 16.O
B PRDEC.S
B PRDEC.O

INTEGER BASIC + *plus* TED II +

TABLE OF CONTENTS

Introduction	2
Programs on this diskette	2
Part I The Integer Basic Plus System	3
Overview	3
Getting Started with Integer Basic Plus	4
Transferring Integer to other Diskettes	5
USER (X) Functions	5
Table of USER (X) Functions	6
DOS ONERR GOTO Function	7
Defining your own Functions	8
Stop List Feature	9
Program Compatibility	9
Part II TED II + Editor/Assembler	10
Assembly Language Whys and Wherefors	10
Additional Reading	11
Background and Features	12
Executive Mode Commands	13
The Edit Module	15
Delimiters	16
Abbreviations	16
Command Syntax	16
Edit Module Commands	17
System Commands	17
Entry Commands	18
Edit Mode Sub-Commands	20
Pseudo-ops	21
Summary of TED Pseudo-ops	21
Group A — Optional 6502 Instructions	21
Group B — TED System Pseudo-ops	21
Group C — Functional Pseudo-ops	23
Addressing Modes and Arithmetic Operators	24
Removing Excess Spaces from the Source File	25
TED Pointers and Memory Map	26
Part III Subsidiary and Utility Programs	27
Programs Relating to Integer Basic	27
Autostart ROM Supplement	27
TED Related Programs	28
Scruncher	28
Universal Disassembler	28
Superdis!	28
Features	29
Method of Use	29
General Comments	30
Compatibility with TED II +	31
How to Handle a Large Program	32
Integer Basic F4 ROM Utilities	
	Inside Back Cover

QUICK REFERENCE LIST - TED II+ COMMANDS AND PSEUDO-OPS

Executive Mode

#	SET DRIVE #
C	CATALOG
D	DISK COMMAND
M	MONITOR COMMAND
L	LOAD SOURCE FILE
S	SAVE SOURCE FILE
O	OBJECT CODE SAVE
A	APPEND FILES
T	TED (ENTER ED/ASM)
Q ←	QUIT

Edit Module

System Commands

NEW	Clear existing source file
LOmem:	Set lower limit of source
HImem:	Set high limit of source
PR#	Same as BASIC PR#
LEN	Display source length
LOAD	Load cassette source
SAVE	Save cassette source
TABS	Set or clear tab stops
List	List lines as specified
Find	Find line # or string
ASM	Assemble source file
Quit	Quit TED and enter EXEC

Entry Commands

Add	Add lines mode
Insert	Insert lines mode
Delete	Delete lines mode
COPY	Copy one range to another
Change	Change string to another
Edit	Edit line #, range or string

Edit Mode Sub-Commands

Carriage rtn

or Ctrl M	Accept entire line
Ctrl Q	Accept to cursor
Ctrl F	Find character
Ctrl I	Insert character
Ctrl O	Insert ctrl char
Ctrl D	Delete character
Ctrl R	Restart Edit
Ctrl C	Exit edit mode

TED System Pseudo-ops

OBJ	Set OBJ address
ORG	Set ORG address
EQU	Assign address to label
AST	Print asterisks
LST ON	Turn on listing
LST OFF	Turn off listing
PAG	Clear screen
PR #	Output to PR#
SYM	Print symbol table
END	End of program

Optional 6502 Instructions

BLT	Same as BCC
BGE	Same as BCS

Functional Pseudo-ops

ASC	Enter ASCII code
DCI	Same except last byte
HEX	Enter hex data
DW	Enter one byte of hex
DS	Reserve bytes
DA	Write label address
DB	Write label adr low

INTEGER BASIC*+

**A RELOCATABLE RAM VERSION
OF APPLE][[®]'s FIRST LANGUAGE**

INTEGER BASIC*

Developed By

JIM McBRIDE

and

GARTH HITCHENS

(c) Copyright 1976 by Apple Computer, Inc.

(c) Copyright 1980 by Apple Pugetsound Program Library Exchange

*Under License Agreement

OVERVIEW

Integer Basic Plus was developed to fill a need for those people who bought an Apple II Plus without the Integer Firmware Card. The concept of a RAM version of Integer Basic was first implemented by James McBride when he successfully relocated it on a friends Apple II + with 32K RAM. It worked great, with the exception that all programs had to be SAVED to or LOADED from cassette tape.

A friend of McBrides, Garth Hitchens, was asked to work out the required DOS modifications to enable programs to be LOADED and SAVED on disk. The next step was to cause the RAM Integer to emulate the protocols of RAM Applesoft, and this was soon accomplished.

At this point RAM Integer Basic could do almost anything the ROM version could do except for certain programs which CALL on routines in the Integer ROM which are not actually a part of the Integer Basic language, and, in addition, the RAM version was not relocatable. After more effort, a relocatable version was developed, followed shortly by a self-relocating version.

RAM Integer Basic had become a reality. Now, in discussion with the directors of A.P.P.L.E. in connection with making this available to the membership, it was decided to enhance the language beyond its original power. First the unused RNDX token was modified to USER and 20 functions such as CHR\$, FLASH and Output Hex were predefined, with another 6 functions available for definition by the user. And then a DOS ONERR GOTO routine was added in to complete the package.

McBride and Hitchens are both in their mid-teens and have more than proven their capabilities as programmers. Their combined teamwork has resulted in the production of another product that may be enjoyed by all A.P.P.L.E. members.

GETTING STARTED WITH THE INTEGER BASIC + SYSTEM

The Integer Basic Plus System has been designed for ease of use. Wherever possible it duplicates exactly the functions of the standard ROM based Integer Basic. It is compatible with the Program Line Editor if PLE is first RUN from Applesoft before entering Integer.

If this is your first experience with Integer Basic, you should buy and study the Integer Basic Programming Manual, published by Apple Computer, Inc. and available from your local dealer,

To start the system up, just insert the Integer Basic Plus System master diskette in your disk drive and follow your normal booting procedure, which is described in the DOS 3.2 manual. For a regular Apple II+, it is a matter of turning the power off, inserting the diskette and turning the power back on, which will result in an autoboot.

After the disk has booted, you will be greeted first by the "Hello" program and the Applesoft prompt () . This is the time to RUN Program Line Editor if you care to.

Also at this time, if you already have Integer Basic in ROM, either on the main board of an Apple II or on a firmware card on an Apple II +, and you wish to use Integer Basic Plus rather than the ROM version, enter BRUN ROM INTEGER. This will confuse DOS to the extent that it will not know you have Integer ROM on board and will allow you to use the RAM version instead.

To Enter Integer Basic Plus, just type "INT" followed by a carriage return and Integer Plus will load, and give you the standard Integer Basic prompt (>), indicating that you now may proceed to LOAD and SAVE programs in the normal manner and use Integer as needed.

CAUTION: You may use the FP command to return to Applesoft, but any attempt to LOAD or RUN an Integer program from Applesoft without first entering an "INT" will not be successful. This may be modified in a later version.

TRANSFERRING INTEGER TO OTHER DISKETTES

The DOS contained on your Integer Basic Plus System master diskette is a unique, highly modified 3.2.1 DOS. It is recommended that you immediately make a back up copy of the System master in the event of its subsequent failure. A copy may be made using the Apple copy program or the Single/Dual Drive Copy program from our diskpak 6. In each case, just follow your usual copy procedures.

In some cases, it may be desirable to transfer the Integer program to another previously initialized diskette. If this is done, it is imperative that the "Update 3.2.1" program included on your Integer Basic Plus System master diskette be used. To do this, type the following commands from Applesoft mode:

] LOAD INTEGER	(on Integer master diskette)
] SAVE INTEGER	(on new diskette)
] BRUN UPDATE 3.2.1	(on Integer master diskette)

When the Update 3.2.1 program BRUNs, you will be prompted what to do. Just type the file name of the hello program you have assigned to the new diskette when requested. After the new diskette has been updated, you may use it just as you previously used the master.

Note that *ONLY* the Update 3.2.1 program from the Integer Basic Plus System master diskette may be used. The same program from the Apple master diskette will *NOT* work because the Apple diskette has only standard 3.2.1. DOS.

USER (x) FUNCTIONS

USER (x) functions are a method of expanding standard Integer Basic with additional commands. There are 20 pre-defined USER(x) commands and an additional 6 may be defined by the user. The USER(x) commands may be entered into a program in one of two ways:

1. 100 PRINT USER(x); (where x represents a number from 0 to 25.)
2. 100 N = USER(x) (where x represents a number from 0 to 25.)

Method 1 is preferable because it is processed more rapidly than method 2. The USER(x) function may be incorporated into a PRINT statement, using semicolons, etc., just as you would any standard PRINT function. An example of printing the word "HELLO" to the screen in INVERSE video would be as follows:

```
110 PRINT USER(1); "HELLO"; USER(2) : END
```

This would change the output mode to INVERSE, print "HELLO" and then restore the output mode to NORMAL. Note that the PRINT command need be used only once.

USER(x) functions 16 and 17, which return a HEX\$ or CHR\$ respectively, may be used as in the following example:

```
120 POKE 1,255: PRINT USER(16): END
```

This would return "FF" to your display. A more detailed explanation appears in the following list of USER(x) commands and their application.

TABLE OF USER (x) FUNCTIONS

- 0 CLEAR TEXT SCREEN. This will clear the text screen to the current output mode. If the current mode is INVERSE, then PRINT USER(0) would return an all white screen area.
1. INVERSE. Sets the INVERSE mode; all text output to the screen will now appear as black characters on white.
2. NORMAL. Returns the video display to the NORMAL (white on black) mode.
3. FLASH. Sets character output to alternate between NORMAL and INVERSE. (See 1 and 2, above).
4. CURSOR UP. Moves the cursor up one space. Enter the following line to test this function: 130 CALL -936: VTAB16: TAB2: PRINT "HEL-LO"; USER(4); USER(4);: END This function supplements VTAB.
5. CURSOR DOWN. Moves cursor down one space. Functions as 4, above.
6. CURSOR LEFT. Moves cursor one space to the left.
7. CURSOR RIGHT. Moves cursor one space to the right.
8. FAST MIXED GR CLEAR. Clears the screen to mixed graphics mode (four lines of text on the bottom) to the color previously set with the COLOR = command.
9. FAST FULL GR CLEAR. Clears the screen to full graphics mode (no text panel) to the color previously set with the COLOR = command.
10. CLEAR TO EOP. Clears the text screen from the present cursor position to the end of the page.
11. CLEAR TO EOL. Clears the text screen from the present cursor position to the end of the current line.
12. SCROLL UP. Scrolls the text screen up one line, filling in at the bottom with a blank line of text. If the video mode were set to INVERSE, the fill line would appear as white. If the video mode was set to NORMAL, there would be no apparent difference. Experimentation with the four SCROLL commands, USER(12) to USER(15) inclusive, will show you how to use these functions to draw INVERSE or FLASHING borders, etc., on your text screen.
13. SCROLL DOWN. Functions exactly as 12, above, except scrolls down with top fill.
14. SCROLL LEFT. Functions exactly as 12, above, except scrolls left with fill on the right.
15. SCROLL RIGHT. Functions exactly as 12, above, except scrolls right with fill on the left.

16. **HEX\$**. Returns a hexadecimal value equivalent to the decimal value previously POKEd into location 1, in the range 0 to 255. Example:
POKE 1,128: PRINT USER(16) would return a hex value of 80.
17. **CHR\$**. Returns to the screen the ASCII character whose value has previously been POKEd into 1. Example: POKE 1,223: PRINT USER(17) would print an underscore "_" to the screen or, POKE 1,225: PRINT USER(17) would print lower case "a" to the screen if you have the Dan Paymar chip and can display lower case. (Else it would print a "I"). Note that unlike Applesoft, the Integer Basic ASCII character set of NORMAL characters is in the range 128 to 255. POKEing 1 with a value less than 128 will result in outputting either FLASH or INVERSE characters.
18. **BASIC ADDRESS**. Stores the high byte of the beginning of Integer Basic Plus in location \$00. This is used internally by Integer Basic Plus, but may prove helpful in patching in routines that call on the Integer ROM.
19. **DOS ONERR GOTO**. This routine, based on one written by Andy Hertzfeld, is a DOS error handling routine that functions in a manner similar to the Applesoft ONERR GOTO. Its purpose is to prevent a disk-based program from aborting when an error is encountered, by instructing the program to jump to a specified line number.

It is enabled by entering a "PRINT USER(19)", POKEing the error handling line number, MOD256 into 10, and POKEing the error handling line number /256 into location 11. When an error occurs, the error type will be found by PEEKing at 7 and the line number of the line that caused the error will be returned in locations 8 and 9. Standard DOS error codes, as shown on Pages 114-115 of the DOS 3.2 manual are used. The following is a short program to demonstrate the use of DOS ONERR GOTO

```

10  REM DOS ONERR GOTO DEMO
20  PRINT USER(19);
30  POKE 10, 1000 MOD 256: POKE 11, 1000/256
40  D$ = "": REM CTRL D IN QUOTES
50  PRINT D$; "CATALOG"
60  GOTO 50
70  REM OPEN DRIVE DOOR TO GENERATE
    A DISK I/O ERROR

1000 REM ERROR HANDLING ROUTINE
1010 PRINT "ERROR TYPE : "; PEEK(7) " ";
1020 PRINT "AT LINE : "; PEEK(8) + PEEK(9) * 256
1030 REM CLOSE YOUR DRIVE DOOR
1040 PRINT: GOTO 50

```

NOTE: Use of the DOS ONERR GOTO routine does not inhibit the printing of DOS error messages, which can be a DANGEROUS practice that in direct mode could conceivably lead to damage of your diskette or program, however, for those that care to, the following two program lines will disable and re-enable the DOS error message. Both lines must be used within the program, with 32767 just preceding or a part of the END statement.


```

32766  ERRMSG=256*( PEEK ( 978 )-256
      *( PEEK ( 978 )>127 ) )+2524: POKE
      ERRMSG,18: REM DISABLE ERRMSG

```

```

32767  POKE ERRMSG,4: END : REM RE-ENA
      BLE ERRMSG

```

20 to 25 inc. Are undefined and may be utilized by users for their own purposes. See the following section to define your own functions.

DEFINING YOUR OWN FUNCTIONS

In addition to USER(0) through USER(19) which are predefined, you may set up your own functions with USER(20) through USER(25). The user-defined functions will allow you to enter jumps to any memory location, indexed by its address in the variable RTN. For the sake of consistency, user defined functions should be defined within the program that utilizes them. A text file, "USER DEFINE" has been established for this purpose.

If you wish to add a function to an existing program, just LOAD the program and EXEC USER DEFINE, set the variable FUN to equal the number to be assigned to the function and set RTN to equal the address of the routine to be called by the function. In the following program example, RTN = -936. As a last step, enter a line in your program to GOSUB 32760.

```

> EXEC USER DEFINE
10  REM PROGRAM TO DEFINE AND DEMO
    USER(20)
20  FUN = 20: RTN = -936
30  REM VARIABLES ARE NOW SET UP
40  GOSUB 32760: REM DEFINE THE FUNCTION
50  REM USER(20) NOW DOES A CALL -936
60  REM NOW LETS TRY IT

100  PRINT USER(20); "THE SCREEN IS
    NOW CLEARED": END
32760 - 32765 (User define routine)

```

In addition to defining the unused USER(x) functions, you may also redefine any of the existing ones to suit your own needs, e.g., USER(3) could be redefined to clear the screen, even though it was originally defined to set the FLASH mode. Note that these changes will not be present when the system is rebooted, but they will be carried from program to program.

STOP LIST FEATURE

Integer Basic Plus also includes a stop list feature. To temporarily halt a program listing, enter a CTRL S. Any other key will resume the listing. A CTRL C will abort the listing altogether and return control to Integer Basic.

PROGRAM COMPATIBILITY

Integer Basic Plus to standard Integer Basic

Programs written in Integer Basic Plus will run on a standard Apple II, provided that the USER(x) function is not used and provided that there are no CALLs to special Integer Basic Plus locations.

Integer Basic Plus to Integer Basic Plus

Programs written in Integer Basic Plus utilizing CALLs to special Integer Basic Plus locations will run only on a system with the same memory size as yours. Programs written using only the USER(x) function will run on any Integer Basic Plus system.

Standard Integer Basic to Integer Basic Plus

Any standard Integer Basic program should run under Integer Basic Plus with the exceptions of programs that reference addresses in the Integer ROM. However, these programs may be modified easily so that they will run properly. A program named "CONVERT" has been in Integer ROM addresses in any binary program, and some Integer programs where the address is in Hex.

To use this feature, boot the Integer Basic Plus System master diskette and RUN Convert. Place the diskette with the program to be converted in the disk drive and respond with the name of the program to be converted when prompted to do so. The process is automatic, and the converted program will be SAVED to disk with the name of the original program followed by the suffix ".CON".

Convert will convert most Assembly Language programs but will convert only those Integer Basic programs in which the ROM address is given in hexadecimal, as in a program using Lam's Monitor String Routine.

Users are cautioned again that any programs utilizing either special addresses or the USER(x) function will NOT run on standard Integer Basic. Therefore these features should not be used in programs for commercial use or for others who do not have Integer Basic Plus available.

TED][+

EDITOR - ASSEMBLER

**FOR THE
BEGINNING OR ADVANCED
ASSEMBLY LANGUAGE PROGRAMMER**

**EXTENSIVELY MODIFIED
AND UPDATED by
KEN SMITH**

(c) Copyright 1980 by Ken Smith

A FEW ASSEMBLY LANGUAGE WHYS AND WHEREFORS

Some of you may ask "What is Assembly Language?" or "Why do I need to use Assembly Language; BASIC suits me fine". While we do not have the space here to do a treatise on the subject, we will attempt as briefly as possible to answer the above questions.

Computer languages are often referred to as "high level" or "low level" languages. BASIC, COBAL, FORTRAN and PASCAL are all high level languages. A high level language is one that usually uses English-like words (commands) and may go through several stages of interpretation or compilation before finally being placed in memory. The time that this processing takes is the reason BASIC and other high level languages run far slower than an equivalent Assembly Language program. In addition, it normally consumes a great deal more available memory.

From the ground up, your computer understands only two things, off and on. All of its calculations are handled as addition or subtraction, but at tremendously high speeds. The only number system it comprehends is Base 2 (the Binary System) where a "1" is represented by 00000001 and a "2" is represented by 00000010.

The 6502 microprocessor has five 8-bit registers and one 16-bit* register in the ALU (Arithmetic Logic Unit). All data is ultimately handled through these registers. Even this lowest of low-level code requires a program to function correctly. This program is hard wired within the 6502 itself. The microprocessor program functions in three cycles. It fetches an instruction from RAM memory in the computer, decodes it and executes it.

These instructions exist in RAM memory as one, two or three byte groups. (A byte contains 8 binary bits of data and is usually notated in hexadecimal (Base 16) form.) Some early microcomputers allowed data entry only through 16 front panel switches, each of which, when set on or off, would combine to form a 16 bit "word". At the next level up, data might be entered directly in hex. This requires an additional program in the computer to break the byte down into its respective 8 bits so that the 6502 may interpret it.

At the next level up (requiring still more programming), the user may enter his/her data in the form of a three character "mnemonic", a type of code whose characters form an association with the microprocessor operation, e.g., LDA stands for "LoaD the Accumulator". The standard Apple II has a built-in mini-assembler that permits simple Assembly Language programming.

But even this is not sufficient to create a long and comprehensive program. In addition to the use of a three character mnemonic, a full fledged assembler allows the programmer to use "labels", which represent an as yet undefined area of memory where a particular segment of the program will be stored. In addition, an assembler will have a provision for line numbers, similar to those in a BASIC program, which in turn permits the programmer to insert lines into the program and perform other editing operations. And this is what TED is all about.

Finally, a high level language such as BASIC is itself an assembly program which takes a command such as PRINT and reduces it (tokenizes it) to a single hex byte before storing it in memory.

Before using this or any other assembler, the user is expected to be somewhat familiar with the 6502 architecture, modes of addressing, etc. This manual is not intended to teach Assembly Language programming. Many good books on 6502 Assembly programming are available at your local dealer; some are referenced below.

* Two 8-Bit registers functioning as a 16-bit register.

ADDITIONAL READING

System Monitor	Apple Computer, Inc.	Peeking at Call—Apple, Vol I
Apple II Mini-Assembler	Apple Computer, Inc.	Peeking at Call—Apple, Synertek 6500-20
Synertek Programming Man.		
Programming the 6502	Rodnay Zaks	Sybex C-202
The Apple Monitor Peeled	Wm. E. Dougherty	A.P.P.L.E.
A hex on Thee	Val J. Golding	Peeking at Call—Apple, Vol. II

Sweet 16 Introduction	Dick Sedgewick	The Wozpak II
Sweet 16 the 6502 Dream Machine	Steve Wozniak	The Wozpak II
Floating Point Package	Apple Computer, Inc.	The Wozpak II
Floating Point Linkage Routines	Don Williams	Peeking at Call—Apple, Vol I
Apple II Reference Manual	Apple Computer, Inc.	

BACKGROUND AND FEATURES

The original TED ASM was written by Randy Wiggington and Gary Shannon. It has been distributed "under the counter" by user groups and individuals under many names and in a variety of versions. Seemingly each person handling it added his own modifications and improvements, thus the present version, known as TED II +, we refer to as "version 698". Version 697 was the one which accompanied the Wozpak II.

We have reviewed as many versions of TED as were available to us, taking what we felt were the best features of each and melding them into the current version. Originally TED had many bugs; at this point we believe we have eradicated most, if not all of them. Significant changes made to this version include the addition of a symbol table generator and cross-reference lister, a much improved TABS routine, compatibility with Program Line Editor, editing facilities which closely duplicate those of PLE's and a completely new front end (executive routine).

TED II +, in its present incarnation, is hoped to be an easy to use assembler for the beginner, and at the same time sophisticated enough to be used by advanced programmers. We believe its edit section to be superior to any currently available assembler for the Apple II. The assembler module is one of the very few around that will directly assemble Sweet 16 code. On a 48K Apple, it can assemble up to about 3000 lines of uncommented source code, or about 6K of object. (super-long files require a bit of extra handling). The only two features lacking is an inability to chain and no macro capability. But then we had to stop someplace.

The current version of TED II + is an Assembly Language program comprising the following modules: Editor, Assembler, Symbol Generator and an Integer Basic EXECutive (menu) module, all of which are incorporated into a single Integer Basic program for loading convenience. TED and its associated modules use memory from \$0800 to \$1FFF. See the TED Memory Map for complete details. Future improvements planned for TED include rewriting the front end (executive) in Assembly Language.

There are two modes of control used in Ted II +, each with its own separate set of commands, one set each for the Executive module (using simple alpha-Numeric characters) and the Editor/Assembler, using alphabetic and control characters. The two control modes are further identified in the next two paragraphs, and completely detailed in the following sections.

The EXECutive mode, signified by the “%” prompt character provides the required interfacing of the editor/assembler to the outside world, otherwise known as DOS or Disk Operating System. It contains the routines that allow the user to switch drives, enter Monitor, etc.

The Editor/Assembler mode, signified by the “:” prompt is the real work-horse that handles all the actual editing and assembling chores.

EXECUTIVE MODE COMMANDS

- #:** SET DRIVE #. This command allows you to specify which disk drive is to be accessed next, and the drive selected will be displayed on the menu screen as the “Current Drive”. All subsequent disk operations will reference the selected drive until such time as it is again changed with the “#” command.
- C:** CATALOG. Displays the disk catalog of the current drive. The current drive in use is displayed on the exec menu and may be changed with the “#” command.
- D:** DISK COMMAND. This command displays a catalog of the current drive and allows direct access to DOS functions and may be utilized for such purposes as renaming and deleting disk files, etc. Unlike other TED disk operations, this command does NOT add the “.S” or “.O” suffix to the file name, hence the user must supply this data. Additional disk commands may be entered without inputting a second “D”. A null input, followed by a carriage return will return you to the menu.
- M:** MONITOR COMMAND. This function allows direct access to examine or change memory, try out routines, etc. Normally control will be returned to EXECutive mode upon completion of the command(s). Should it not, to reenter the assembler module from Monitor, type “803G” and a carriage return.
- L:** LOAD SOURCE FILE. This command will ask you to indicate the name of the file to be LOADED. If you cannot recall, or are unsure of the spelling, hit return and a catalog will be displayed. At this point the file name may be input or a carriage return will abort the LOAD routine and return you to the menu. The name of

this file will now be displayed as the "Current File" on the menu panel, and you will be placed in TED (edit/asm) mode for editing or assembling the source file. You should also remember that EXEC automatically attaches the ".S" suffix to the file name, so it should *NOT* be entered by the user.

An additional feature of the LOAD command allows you to specify an alternate area of memory in which to LOAD your source. (Default LOADING starts at 8192 decimal.) When prompted for the file name of the file to load, add the at sign at the end e.g.; "TEST@". Exec will then ask you for the decimal starting address for the source file. This may be at any memory address between 8192 and 32767 decimal. This feature allows greater flexibility in assembly of routines that would assemble in areas where the source file normally resides.

CAUTION: *No check is made to determine if the source will fit into the remaining available memory, thus it is possible to overwrite the EXEC variables, the EXEC program itself, and DOS as well.*

S: SAVE SOURCE FILE. This command SAVES to disk the current source file in memory under the file name provided by the user. If a carriage return is input on the initial prompt for the file name, the disk catalog will be displayed and you will be prompted a second time for the file name. If a second carriage return is entered, the SAVE routine will abort and return you to the EXEC menu. As in the LOAD command, the ".S" suffix is automatically added by the EXEC program and should *NOT* be input by the user. The "Current File" as displayed on the menu will be updated to reflect the name of the file just SAVED.

O: OBJECT CODE SAVE. This command SAVES to disk the object code created by the most recent assembly. This object code will be SAVED under the name of the "Current File" as displayed on the EXEC menu, along with a ".O" suffix. the DOS BSAVE command will be printed on your screen, along with the Address and Length parameters, so that you may record these addresses for future reference. If there is no "OBJ" pseudo-op in the source, then the object file will be assembled at and SAVED from the 48K default address of \$7000. The OBJECT CODE SAVE command is disabled when TED is first run, prior to an assembly, after a file LOAD, or if TED II + is reentered with a RUN TED command, rather than GOTO TED.

CAUTION: *The OBJECT CODE SAVE command will save the LAST contiguous block of machine code created by the assembler, so that if more than one "OBJ" in the source file is encountered by the assembler, only the block of code (starting at the address specified by the LAST "OBJ" pseudo-op found) will be saved.*

A: **APPEND FILES.** This routine allows you to combine any two source files existing on diskette. The first named file will be placed lowest in memory and the source line numbers will continue in sequence as the second file is APPENDED in. Any source file in memory when the APPEND command is entered will be **OVERWRITTEN**.

When the APPEND operation has been executed, you are given the option of SAVEing the APPENDED file to disk. If this is selected, you will be asked to enter the file name of your choice (without the ".S", of course). If a carriage return is entered without specifying a file name, then the file will be SAVED under the name of the first of the two files APPENDED, plus a suffix "-A". Again the "Current File" display on the menu will be updated to reflect the name chosen, or will remain unchanged if the SAVE was not executed. The APPEND function will abort if a carriage return is entered as *EITHER* the "original file" or the "file to append". Upon completion of the APPEND function, control is transferred to the TED mode.

When files are APPENDED, often a number of EQU statements are located in the middle of the source file. They must be moved to the beginning of the source file for the assembler to work properly. If this is not done, the assembler will assign the wrong offset to Branch instructions. (When these EQU's are copied to the beginning of the source file, remember that you must delete those lines that are still in the middle of the source.) See COPY Command.

T: **TED (ENTER ED/ASM).** This command transfers control from the Exec Module to the TED Editor/Assembler module. The TED Module has its own set of commands for editing and other functions which are described in detail in the next section.

Q: **QUIT.** This command exits TED altogether and returns you to Integer Basic. As displayed at the top of the screen, you may reenter TED with a "GOTO TED" command, followed by a carriage return to restart the program with all variables and files intact. "RUN" may also be used, but the "Current File" will be lost and the OBJECT CODE SAVE function will be disabled until such time as an assembly is again made. In addition, the current drive will also be reset to one. In either case, the current source file in memory will *NOT* be lost, and may be SAVED or edited as desired.

THE EDIT MODULE

Here is an editor you can really look forward to using with pleasure. Its great power is in the ease of entering commands, yet with the capability of combining multiple commands and multiple ranges within a single input. For example, it is possible to enter a command such as the following:

CHANGE 15,63/70,79"HELLO"GOODBYE"

The effect of this command will be to change the string "Hello" to read "Goodbye" within the two specified ranges. If "Hello" is found in lines numbered 15 to 63 inclusive or in lines numbered 70 to 79 inclusive, it will be changed to "Goodbye". If "Hello" appears in lines outside the specified parameters, it will be unaffected. Wherever a multiple range is applicable to a command, it may be used, and command syntax will be discussed more fully under the data for the individual commands.

This section covers commands used in the Edit Module (TED mode). There are two forms of command syntax: those that establish a specific function, such as INSERT or LIST, and those which are used under the control of the EDIT command, such as CTRL I (Insert character) or CTRL Q (Accept line). The former are composed of one or more alphabetic characters, while the latter are control characters.

DELIMITERS

Three types of delimiters are used. The comma "," is used to delimit a range, as in LIST 112,200. All lines between 112 and 200 only will be listed. The slash "/" is used to delimit two ranges, as in LIST 112,200/300,347. In this instance all line numbers falling between 112 and 200 and between 300 and 347 will be listed, but not those less than 112, nor those greater than 347, nor will lines 201 to 299 be listed. Up to 12 multiple ranges may be specified in a single command. The third delimiter is the double quote (") which serves to delimit strings. If only one string name is required, as in FIND "HELLO", then the string will be enclosed in quotes. If two strings are references, as in CHANGE "HELLO"GOODBYE", only one double quote will be used to delimit the end of the first string and the beginning of the second string.

ABBREVIATIONS

The following symbols and abbreviations will be used to demonstrate correct syntax usage under the individual command names:

<line #>	a single line number
<range>	<line #>, <line #>
<m/range>	<range> / <range> or <range><line#> or <line #> / <line #>
<string>	a literal (or alphanumeric) string

COMMAND SYNTAX

The Edit Module commands are composed of single or multiple alphabetic characters. Certain commands, such as LIST, may be used by entering "LIST" OR by abbreviating to "L". Commands which may be abbreviated are shown in the following table with the *OPTIONAL* characters printed in italics, as in: *List*. Where no italics are printed the *ENTIRE* command name must be used. Each command must be terminated by a carriage return.

EDIT MODULE COMMANDS

SYSTEM COMMANDS

- NEW** Deletes the existing source file and resets the pointers. It does NOT reset memory limits.
- LOmem:** <decimal address>
Sets the boundaries for lower memory to be used by TED II +, which is normally the start of the source file, decimal 8192 by default. If it should be set lower than 8192 it will overwrite some or all of the symbol cross reference routine, the editor buffer and assembler. See TED II + memory map. LOMEM: also performs an automatic NEW, thereby erasing any source file in memory.
- HImem:** <decimal address>
Sets the highest memory limit available to Ted II + for source files and the symbol table, (which is created by the assembler.) Care is urged in resetting HIMEM:, as it can overwrite the Integer Basic EXEC program and destroy DOS as well. Upon initial entry into Ted, HIMEM: is automatically set for the maximum available memory.
- PR #** (1,7)
Same as BASIC PR#. Quitting to EXEC mode resets to PR#0. The purpose of this command is for outputting hard copy listings of the source file. Its use is limited to output devices which are interfaced in peripheral slots 1 to 7. PR# is also a "pseudo-op" and is discussed in that section.
- LEN** Displays the current size of the source file in memory and the number of available bytes remaining for the source file.
- LOAD** This command will load a TED source file from cassette tape and is used from TED mode. A disk LOAD must be performed from the EXECutive module.
- SAVE** This command will SAVE a TED source file to cassette tape and is entered from TED mode. A disk SAVE must be performed from the EXECutive module.
- TABS** <column> / <column> / <column> " "
Each time TED is entered, tab stops are set to a predetermined value for LISTing the source file. The TABS command is designed primarily to temporarily change the tab stops for a printed listing of the source file. Column is a decimal number specifying the horizontal tab position of each field of a line of source listing. The start of each field is defined by a space. An asterisk "*" is used for comment lines and any line containing one or more asterisks will not be tabbed. Tabbing is used in TED II + to

provide formatting of listings without the use of extra spaces, which are highly wasteful of memory. Therefor in entering lines, no more than one space should be used between fields.

List <line#>
List <range>
List <m/range>
List <m/range> / <m/range>

LIST the line #(s) specified. If no <line #> or <range> is specified, then the entire source file will be LISTed. LISTing may be controlled with the space bar for examination. The LISTing will advance one line each time the space bar is depressed. Any other key will cause the LISTing to contine at normal speed. A LIST may be aborted at any time with a CTRL C.

Find "<string>"
Find <range>"<string>"
Find <m/range>"<string>"

Locates and prints all lines in which <string> occurs. If <string> or <m/range> is specified, then only those portions of the source will be searched. Certain substrings with sequential characters will not be found. "AAB" of "AAAB" will not be found.

ASM This is the command that directs TED to actually commence the ASseMbly process. If the LIST option is on, then ASseMbly will halt at the line preceding a line in which an error has been encountered. The space bar will also halt ASseMbly in the same manner. In either case, ASseMbly may be resumed by hitting any other key. A CTRL C will abort this routine at any time. Errors detected while in LIST OFF mode will be displayed but will not halt the ASseMbly.

At the conclusion of ASseMbly, TED will report in decimal both the number of errors detected and the total bytes generated. Entering a NOP as the last line of your program may prove helpful in determining the address of the last byte ASseMbled.

QUIT QUITs TED mode and returns you to the EXECutive. Does not QUIT the program.

ENTRY COMMANDS

Add Puts the user in the "ADD lines" mode. If a source file is already resident in memory, then ADD will start with the next available line number. If there is no source file in memory, numbering will commence with line 1. ADD is terminated with CTRL C as the first character of a line, followed by a carriage return.

Insert <line #>
INSERT functions in a manner similar to ADD, but its function is to place program lines in between other lines of source. Lines entered in this mode will be placed immediately before the

<line #> specified in the command, and all lines following the INSERTed lines will automatically be renumbered. Line(s) may be INSERTed before any existing line. INSERT is terminated with CTRL C as the first character of a line, followed by a carriage return.

Delete

<line #>

Delete

<range>

Delete

<m/range>

DELETE is the inverse function of INSERT, in that it removes unneeded lines from source file memory as specified by the entry parameters. Because all lines following the deletion(s) will automatically be renumbered, *IT IS IMPERATIVE* to do deletion(s) from highest to lowest line numbers, whether it is a line at a time or a <range> or a <m/range> otherwise the wrong lines will be DELETED because of the renumbering process.

COPY

< range> TO <line #>

Notice the use of the special delimiter "TO" for this command. COPY will COPY a <range> of line numbers and INSERT them immediately before <line #> in the source file. As in INSERT, all line numbers *FOLLOWING* the INSERTion will automatically be renumbered. A single line to be copied must be treated as a <range>, i.e.; "COPY 32,32 TO 27". *SPECIAL NOTE:* The original lines which were copied (and by now may have been renumbered) will *NOT* be deleted. This must be handled as a separate operation by the user.

Change

" <string¹>"<string²>"

Change

<range>"<string¹>"<string²>"

Change

<m/range>"<string¹>"<string²>"

One of the more powerful features of the Edit Module, CHANGE permits the selective substitution of <string²> for <string¹>. When the CHANGE command is entered, you will be asked if "ALL OR SOME (A/S)" occurrences of <string¹> are to be changed. If ALL is selected, <string²> will be substituted for all occurrences of <string¹> within the source file, or, if a <range> or <m/range> is specified, only those occurrences within the <range> or <m/range> will be changed.

The SOME option allows you complete control of the substitution process. It will display on the screen each line in which <string¹> occurs, showing the substitution to <string²>. Hitting the ESC key at this time will cancel the CHANGE for this one line, and will permit processing to resume to the next line in which <string¹> occurs. Entering any key *OTHER THAN* ESC will cause the line to be CHANGED as it appears in the display, and processing will resume. In either case, if no <range> or <m/range> is specified, the entire source file will be processed. Be sure to enter the double quote marks as shown in the preceding examples.

Note that a substring may be changed with this command. For example, if you desired to change the string "ABC" to "XYZ" and did not control the substitution process with the "SOME" Option, then the word "ABCDE" would be changed to "XYZCE". To eliminate surprises like this, it is wise to choose the "SOME" option and control the CHANGES. Certain substrings will not be found as explained in the FIND command.

Edit	<line #>
Edit	<range>
Edit	<m/range>
Edit	"<string>"
Edit	<range>"<string>"
Edit	<m/range>"<string>"

EDIT is the most powerful of the TED module commands. Its parameters allow for not only EDITing a <line #>, <range> or <m/range>, but in addition its syntax permits the entry of a "<string>", <range> "<string>" or <m/range>"<string>", thus permitting EDITing of a line whose number is not known but which is known to contain the string "<string>". The EDIT command has its own self-contained set of sub-commands, all of which are Control characters, and are fully explained in the next section.

When EDIT mode is entered, each line to be EDITed is displayed on the screen *WITHOUT* tab stops set, and the cursor is placed at the beginning of the line to be EDITed. The forward and back arrow keys may be used to position the cursor at the point where a change is to be made. In addition, the CTRL F (FIND) sub-command may be used to find a specified character in the line. All of the sub-commands resemble in function those used by Program Line Editor.

EDIT MODE SUB-COMMANDS

Carriage

Return Accepts the entire line as it appears on the screen.

CTRL O Accepts only that portion of the line up to, but not including the cursor.

CTRL F Finds the first occurrence of the next character entered after the CTRL F and positions the cursor on the preceding character.

CTRL I Inserts character(s) into the line, beginning at the cursor position. Characters, including spaces, will continue to be inserted until such time as either the forward arrow key or the carriage return is hit.

CTRL D Deletes the character under the cursor.

CTRL O Overrides the other sub-command control characters and functions like CTRL I by allowing a Control character to be inserted into the line at the cursor position. Unlike CTRL I, CTRL O inserts only a single control character with each use. It does *NOT*

work with a CTRL M (carriage return). Inserted Control characters are *NOT* displayed *EXCEPT* when the cursor is positioned over a Control character. Uses for this function will best be found by experimentation.

- CTRL R** Restart EDIT of current line. Any changes made in the current line being EDITed prior to the entry of the CTRL R will be removed and the line will be rewritten to appear as it did when first entered, with the cursor over the first character.
- CTRL C** Cancel EDIT mode. This is an unconditional exit from EDIT Mode and may be issued at any time or position while in EDIT mode. A carriage return is *NOT* required.

PSEUDO-OPS

The 6502 microprocessor is programmed with an instruction set of 56 three character opcodes which are recognized by the TED II + Assembler. In addition, TED is one of the few available assemblers that will recognize and assemble the 26 opcodes of Steve Wozniak's Sweet 16 simulated 16 bit microprocessor.

A good assembler must have the ability to recognize and interpret its own set of "PSEUDO-OPCODES", which are a convenient means of instructing the assembler how to process certain parameters or data that the user may need to specify.

TED's pseudo-ops may be grouped into three categories: Group "A", which is composed of two pseudo-ops offered as options to standard 6502 instructions, Group "B", which pertain to TED system functions, and Group "C" which offer the programmer greater freedom and facility in actually entering data into the source file. They are discussed in the following summary.

SUMMARY OF TED PSEUDO-OPS

GROUP "A" OPTIONAL 6502 INSTRUCTIONS

- BLT** Branch Less Than (optional form of BCC — Branch Carry Clear).
- BGE** Branch Greater than/Equal to (optional form of BCS — Branch Carry Set).

GROUP "B" TED SYSTEM PSEUDO-OPS

- OBJ** \$<address>
OBJ specifies where in memory the OBJECT code created by the assembly process is to be stored. The purpose of using OBJ is to prevent storing the code in an area where it could overwrite TED or other required data. It may prove useful to store the OBJECT code at a location that is an even offset from the address where the program is designed to *RUN*, e.g.; you might use an OBJ address of \$6800 for a program with an ORG of \$0800.

ORG

\$<address>

ORG stands for ORiGinate and specifies that area of memory where the assembled program will RUN. Often the program cannot be stored at the location where it will later run, owing to conflicts with other programs (such as TED) currently resident in memory. The solution is *NOT* to assemble with a ORG address of \$6800 for a program to run at \$0800, since it then would assemble with addresses that would be incorrect when moved to \$0800. Simply moving it is not the answer.

Instead, by setting "OBJ \$6800" and "ORG \$0800", the assembler will create and store the code starting at \$6800, *BUT* it will be written with the correct addresses to *RUN* at \$0800. When the OBJect code has been stored at the desired location, you may use the OBJ SAVE routine of the EXECutive to save onto disk. You may move the SAVED code to its proper location in either of two ways: you may BLOAD the OBJect code and use the Monitor's built in memory move routine and then BSAVE it at the new address, or you may BLOAD the OBJect code directly to the new address by using the A\$ parameter with the BLOAD command and then BSAVE it in the usual manner. You should be certain you have previously SAVED the source file before moving the OBJect code, as this could overwrite and hence destroy either TED or the source file itself.

EQU

<data>

The EQU operator assigns a value (normally an address) to a label. Once a label has been defined, that label can be used in the source file. For example, the address of the BELL routine in the APPLE monitor is \$FF3A. If we enter a line "BELL EQU \$FF3A", any time that we want the routine to sound the bell in the APPLE, we simply need to enter a line such as "JSR BELL". The assembler will search its symbol table and find the address assigned to 'BELL' and write the code as needed. If no '\$' is used, the number is assumed to be less than 256 (decimal) and to be a decimal number, which will be converted to HEX by the assembler. TED does not differentiate between zero page EQUates as some other assemblers do. Some assemblers will use the Pseudo-op "EPZ" when EQUating zero page locations.

AST

<hexadecimal number>

During assembly, prints the number of asterisks as specified. This is useful for separating sections of the program or formatting title and comment lines.

LST ON

Turns the assembly listing on wherever specified, to screen or printer. May be used anywhere and as often as desired within the source file. Set by default when entering TED.

- LST OFF** Turns the assembly listing off wherever specified, to screen or printer. May be used anywhere and as often as desired within the source file.
- PAG** Clears the screen when encountered during assembly. If assembly output is to a printer, PAG will send a FF (form feed) to the printer.
- PR#** <slot>
PR# \$<address>
 Sends assembly output to the slot or address specified. If \$<address> is used, then output will go to \$<address> where hopefully you will have loaded a printer driver! For example "PR# \$300" will send output to a routine that resided in memory at \$300.
- SYM** Causes the assembler to produce a Symbol-Cross Reference Table at conclusion of assembly. The symbol table consists of all of the label names (beginning with an alphabetic character) in near alphabetic order, along with the address assigned to that label by the assembler, and all source file line numbers in which that label appears. SYM may be placed in the source file at any point. SYM will print only those labels beginning with an alphabetic character. Control of the symbol table listing routine is the same as control of the LIST or ASM process, i.e.; listing may be temporarily halted with the space bar, resumed with any other key and aborted with a CTRL C. Following a CTRL C, or at the conclusion of the symbol table, you will be returned to the editor (TED mode).
- END** This is an optional operator that flags the last line of source to be assembled. If target labels are located beyond END, undefined label errors will occur. END is supported by the assembler, but not required in any file.

GROUP "C" FUNCTIONAL PSEUDO-OPS

ASC '<string>'
ASC "<string>"

This handy pseudo-op will write the ASCII code for the characters contained in <string> directly into memory. There are two delimiter options available. The single quote (') will write the ASCII code with the high bit clear (standard ASCII as recognized by Applesoft). The double quote (") will write the ASCII code with the high bit set (negative ASCII as recognized by Integer Basic and Monitor). The standard ASCII character set uses \$00 to \$7F; the negative ASCII character set uses \$80 to \$FF.

DCI '<string>'
DCI "<string>"

DCI performs the identical function as ASC, with the exception that the *LAST* character written will have the high order bit set or cleared exactly opposite to the preceding characters in the string, as was determined by the choice of delimiters " or '. For example, in the string 'HELLO', the HELL would be written in standard ASCII and the O in negative ASCII.

Both the ASC and DCI operators will recognize a maximum of 39 characters on one line; additional characters will be ignored. In addition, during assembly, while ASC and DCI will be assembled correctly, the OBJECT code portion will show only the address and OBJECT code of the first three bytes.

HEX <data>
Functions like ASC but writes hex bytes into memory. <data> is written as "003AFF8C" (no quotes) with no spaces between bytes. During assembly only the address of and the first three bytes will be shown in the OBJECT code listing.

DW <expr>
Writes a one byte HEX equivalent of <expr> into memory. <expr> may be written in either decimal <expr> or HEX \$<expr>.

DS <expr>
Reserves the number of bytes specified by <expr> in memory, and is generally used with a preceding label. <expr> may be written in either decimal <expr> or HEX \$<expr>.

DA <label>
Writes the address of <label> into memory low byte first, then high byte. May be modified with the +, -, or * arithmetic operators.

DB <label>
Writes only the low order byte of <label> into memory. May be modified with the +, -, or * arithmetic operators.

ADDRESSING MODES AND ARITHMETIC OPERATORS

The SET operator (SWEET 16) will not recognize an immediate mode operand. For example, SET R1, \$0300 requires that R1 be used as a label assigned to 1, and that \$0300 be assigned a value of \$0300. Therefore, the statements R1 EQU 1 and \$0300 EQU \$300 must be entered at the beginning of the source file. Additional information on SWEET 16 may be found in "Docupak Vol. I: The Wozpak II".

The crosshatch “#” is used to indicate an immediate mode operand. The following are some examples of its use:

#<LABEL	—Low byte of LABEL address
#>LABEL	—High byte of LABEL address
#' <chr>'	—ASCII of <chr> with high bit clear
#" <chr>"	—ASCII of <chr> with high bit set
#<num>	—Decimal number (0 to 255)
#\$<num>	—Hex number (0 to FF)

Operands may be modified with the arithmetic operators +, -, and *. These are single byte modifiers only. For example LDA LABEL-1 is a legal statement and would result in LoADing the Accumulator with the data found in memory at the location specified by LABEL, less one byte.

REMOVING EXCESS SPACES FROM THE SOURCE FILE

Because the TABS operator automatically adds spaces as required for LISTing and ASseMbly, there is no need to enter extra spaces in the source file for formatting purposes. To do so consumes huge chunks of precious memory.

Since earlier versions of TED II did not have the TABS automatically implemented, many routines have been written using extra spaces to provide formatting of lines. If one of these files is loaded into TED II +, the formatting will not be correct, since each space encountered in the listing causes a tab to the next tab field. These files can be “processed” to remove these extra spaces, which will also free up much memory. By using the CHANGE command, the editor can go through the source file and remove these spaces.

To accomplish this file processing:

Enter the command CHANGE” (A) ” (B) ”. (A) should be 2 to 6 spaces. (B) is a single space. By choosing the “ALL” choice of the CHANGE command, most of the spaces will be removed on the first pass. Subsequent passes should direct the editor to change 2 spaces to one space. It may take 2,3 or more passes to fully process the file. To check that all the extra spaces have been removed, use the FIND command and search for 2 spaces. If none are found, then the source file can then be saved to disk.

If the source file contains comment lines or title boxes, the user must do these changes selectively, using the <range> capabilities of the CHANGE command.

TED II + will LOAD the file of an earlier TED at the proper address, and a subsequent SAVE of the file will save it with the proper address pointers. Editing of the source file created on an earlier version of TED II may be occasionally required.

If the source file is saved directly by the user, using the BSAVE command, the user must add the ‘S’ suffix of the file name, so that TED II + can find it later.

TED POINTERS

TED MEMORY MAP

Source file pointers are located as follows:

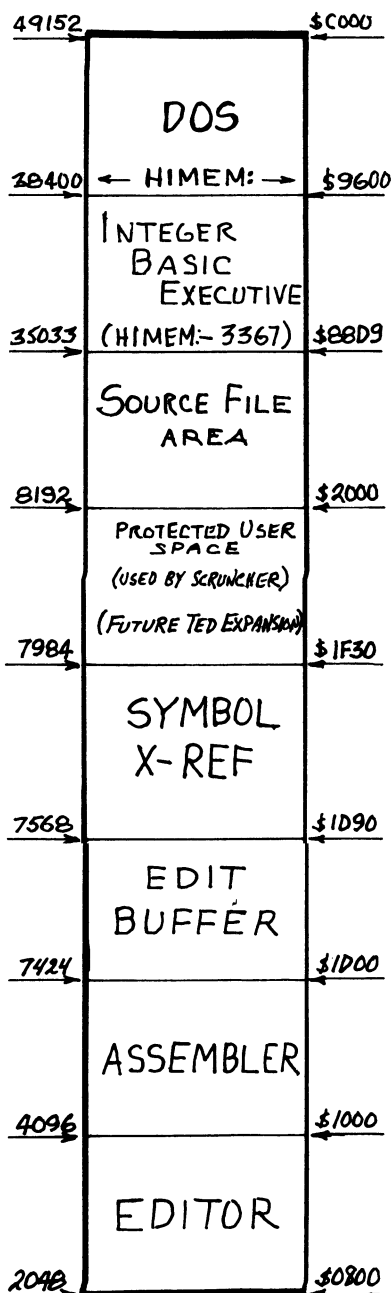
- \$A & \$B** LOMEM: and start of source file
- \$C & \$D** HIMEM: Last available memory address, less one.
- \$E & \$F** Address of current "END OF SOURCE".

By using these addresses, the user can still SAVE the source file to tape or disk even if the EXEC program (which resides below BASIC HIMEM:) is accidentally overwritten.

TED II + CREDITS

The following individuals, in varying degrees, have all contributed to what we call Version 698 of TED II +. We would like to offer our thanks to each. For those we may have overlooked, apologies.

Darrell and Ron Aldrich
 Val Golding
 Andy Hertzfeld
 Neil Konzen
 Pete Rowe
 Gary Shannon
 Ken Smith
 Peter Soule
 Wayne Throop
 Steve Wheeler
 Randy Wiggington



PART III – SUBSIDIARY AND UTILITY PROGRAMS

PROGRAMS RELATING TO INTEGER BASIC

This is a collection of four functions and a utility that can further enhance the capabilities of Integer Basic or Integer Basic Plus. They are all written in Integer Basic and can either be LOADED from disk as you start to write a new program, or they may be LOADED from disk and an existing Integer program can be APPENDED in.

To APPEND a program to one of the function programs, follow these easy steps:

1. LOAD (function program)
2. Enter the following direct commands from the keyboard:
HML=PEEK(76)
HMH=PEEK(77)
POKE76,PEEK(202)
POKE 77, PEEK(203)
3. LOAD (Integer program to be appended)
4. Enter the following direct commands from the keyboard:
POKE 76, HML
POKE 77, HMH

And the two programs will now both be resident in memory. Each of the Function utilities are sufficiently documented in REM lines to explain their operation.

Each of the four function utilities is used to simulate in Integer Basic or Integer Basic Plus, a routine that is part of the Applesoft language. CHR\$ will print the ASCII character of the value used. STR\$ will return in the form of a string, the value input. VAL will return a variable equal to the value of the string input. String Array allows the user to define and use a string array of from 1 to 99 elements.

The fifth program "Integer Basic Post-Editor" first appeared in Call – Apple, March-April, 1980, and is documented there. Its purpose is to allow you to enter "illegal" program lines into an Integer Basic program, i.e.; lines containing commands like HIMEM: or CLR.

AUTOSTART ROM SUPPLEMENT

The "SUPPLEMENT" consists of a group of utility routines by Steve Wozniak that reside in the F4 and F8 ROMS of a standard Apple II. They are not present in the Apple II + nor in an Apple II that has had the "old" Monitor ROM replaced with the Autostart ROM. Instructions for using Single Step, Trace and Mini-Assembler can be found in the Apple II Reference Manual. Wozniak's programs have been modified for "Supplement" by Guil Banks who, in addition, has written the included "Convert" program.

The Supplement may be entered from Monitor via the CTRL Y function and requires that your Hello program contains lines such as:

```
100 HIMEM: -29440: REM $8D00
110 POKE 1016, 76: POKE 1017, 0: POKE
    1018, 141 : REM SET JUMP TO $8D00
    AT $3F8.
```

The Supplement's prompt character is a ")" and all Monitor commands can be entered from within the Supplement, except that the Mini-Assembler is enabled by entering a CTRL A. A slash "/" will exit the Mini-Assembler and return you to the Supplement. The "Trace" function also uses the slash to terminate. A CTRL S entered while in Trace mode will cause a pause until another key is hit.

The "Convert" routine will convert a hexadecimal value in the range of 0-FFFF to its decimal and twos compliment decimal equivalents; it will also convert a decimal value from 0 to 65535 to its hexadecimal and twos complement hex values. Convert is entered with a CTRL T and exited with a slash. Its prompt character is the carat "^".

To set decimal to hex conversion enter "^T carriage return"

To set hex to decimal conversion, enter "^H carriage return"

Convert will remain in the conversion mode set by the T or H parameters until changed or until a slash is entered. Note that control is returned to the Monitor when any function other than those in the supplement is used. Negative numbers are not required or allowed. Conversion examples are below.

"^T carriage return ^936 carriage return " will display: 03A8 FC58

^H carriage return ^FC658 carriage return will display: 64400 -936.

PART III – SUBSIDIARY AND UTILITY PROGRAMS

TED RELATED PROGRAMS

SCRUNCHER

"Scruncher", by Ken Smith is a Sweet 16 program whose sole purpose in life is to "scrunch" out unwanted spaces in TED source files. It should be BLOADED from TED's EXEC, using the DISK command and then, using the MONITOR command, enter a "1F50G", and it will proceed merrily on its way. Scruncher requires only that you have a source file already in memory, and, upon completion of its chores will return you to the TED EXECutive. Scruncher is also built into Superdis!, the companion disassembler for TED II +.

UNIVERSAL DISASSEMBLER/SWEET 16 TRACE PROGRAM

This program by Peter Soule has been included to balance out the collection. Since we have an assembler that assembles Sweet 16 and Sweet 16 itself, we jumped at the chance to disassemble and trace Sweet 16 on the same diskette. Universal Disassembler does just that. It does not create a TED source file as does Superdis!, but it does handle Sweet 16, and it can be output to the printer so that you have a hardcopy disassembly that can later be made into a source if desired.

The instructions are short and "sweet". Just BRUN UNIVERSAL DISASSEMBLER or, if you want hardcopy, BLOAD it, enable your printer and CALL 2048 from Integer Basic or enter 800G from Monitor. Universal Disassembler uses the "old" monitor ROM extensively, so Integer Basic in ROM is a prerequisite to its use. Its prompt is the dollar sign "\$".

Only two commands are required for disassembly, which functions in a manner very similar to the Apple II disassembler. For disassembly commencing at address \$hhhh, enter:

HHHHL (carriage return)

to produce one screen page of disassembly listing. In this mode, Universal Disassembler will automatically jump between 6502 and Sweet 16 code each time it encounters a JSR SWEET16, JSR SWEET16+03 and will return to 6502 mnemonics whenever the 00 "RTN" code is encountered. It cannot handle a return to 6502 via a branch operator.

To force disassembly into Sweet 16 code irrespective of the language the code is actually written in, enter:

HHHHS (where HHHH is the address to commence disassembly)

A carriage return will return control to the Monitor with a beep.

To trace Sweet 16 you must replace each call to SW16 or SW16+03 with a JSR to \$09C9, e.g.; 20 89 F6 - 20 C9 09. Run the program from Monitor at its normal entry point. During the interpretation of Sweet 16, you may enter an "S" to single step through the Sweet 16 portion only. During execution of native 6502 code, the program will execute normally without trace.

SUPERDIS! by Steve Wheeler

This program provides a companion disassembler for the TED II + editor/assembler provided on this diskette. It can be easily modified to work with other versions of the TED assembler. Using a machine language program as input, the disassembler will interact with the user to produce an assembler source file which can be reassembled with a minimum of editing.

FEATURES:

The disassembler handles 6502 code, and directly supports the following pseudo-operations:

HEX	for hexadecimal data
ASC	for ASCII data
DCI	for ASCII data delimited by the last character in the string
DA	for 16-bit data words.

Since the TED II + assembler uses single and double quotes to specify whether the high order bit of the characters in an ASCII string is to be cleared or set, respectively, the user has the option of using single or double quotes when handling data with the ASC and DCI pseudo-ops.

Upon completion of file building, the source file can be processed to remove unneeded labels, and to convert operands which refer to a label into that label.

During the building of the source file, the program checks after each character is output to determine if the source file has reached HIMEM. If so, then the line being entered is deleted from the file, and control is returned to the user.

METHOD OF USE:

Running the program (BRUN DUPERDIS from disk, or 1000G if the program is already in memory or being loaded from cassette) will set the control-Y hook. The disassembler is then available to the user. In the following list of commands, (Y) is used to signify a control-Y, and (R) signifies a carriage return.

COMMAND	ACTION
xxxx,zzzz (Y) (R)	Initial entry into the pgram. File pointers are initialized, the ORG of the program being disassembled is set, and from xxxx to zzzz is disassembled as 6502 code.
xxxx,zzzz(Y)'A(R)	The pseudo-op ASC is used to specify an ASCII string. Single quotes are used as the string delimiter. Data is output as a continuous string from xxxx until zzzz or a non-alphanumeric character is reached.
xxxx.zzzz(Y)"A(R)	Same as the previous command, but with double quotes used as the delimiter.
xxxx.zzzz(Y)C(R)	6502 code is disassembled from xxxx to zzzz. File pointers are not initialized. The ORG pseudo-op is not output again. Any invalid instructions are handled as single data bytes with the HEX pseudo-op.
xxxx(Y)'D(R)	The pseudo-op DCI is used to specify an ASCII string beginning at address xxxx. Single quotes are used as the string delimiter. Data will be output as a continuous string until an alphanumeric character with the high-order bit set or a non-alphanumeric character is reached.
xxxx(Y)"D(R)	Same as previous command, except double quotes are used as the string delimiter, and output stops with an alphanumeric character with the high-order bit clear or on a non-alphanumeric character.
xxxx.zzzz(Y)H(R)	The contents of memory from xxxx to zzzz are treated as data. The HEX pseudo-op is used, with up to eight data bytes per line. Data is output such that the last character of the label field for all data lines after the first will be either an '8' or a '0'.

xxxx(Y)W(R)

The contents of xxxx and xxxx+1 are output in reverse order following the character string "DA \$".

(Y)*(R)

Commences file processing. All labels which are not referenced by the operand of an instruction in the file are removed. All operands which reference a label are converted to the appropriate label. Zero page and absolute references to addresses outside the program being disassembled are not disturbed.

If the segments of the program being disassembled are contiguous, that is, if there are no wasted bytes between them, it is possible to drop the xxxx from a command. You *MUST* specify xxxx for initial entry into the disassembler, when skipping a section of the program being disassembled, or when returning to the disassembler after using the TED editor to modify the source file being built.

GENERAL COMMENTS:

Labels consist of an "L" followed by the four hex digits of the address.

Not all operands which reference addresses within the program being disassembled will be converted to labels. This will most often occur when a large hex data section exists, and an absolute or absolute indexed load or store references an absolute address which is not at the beginning of a HEX data line, and is therefore not associated directly with a label. This can be taken care of by editing after the file is built, or by dividing the hex data into two or more segments during disassembly, if the situation is known beforehand.

The disassembler does not directly support entry of comments or pseudo-ops not in the command list. However, since the disassembler is designed to reside in memory between the assembler and the text file, it is a simple matter to enter the editor (803G), insert any desired line or lines, then press RESET and resume operation of the disassembler. The only limitation is that if the first character of a manually entered line is an "L", the first five characters of the line will be treated as a label during file processing. Any line which starts with a character other than "L" is assumed to be comment line, and is skipped by the label matching and removing algorithms.

The larger the source file, the longer processing the file will take. If the size of the source file is doubled, the processing time will increase by about a factor of four. The following table gives a few approximate times for source files of various sizes.

FILE LENGTH (lines of code)	PROCESSING TIME (approximate)
100	7 seconds
200	27 seconds
250	43 seconds
500	2 minutes, 50 seconds
1000	11 minutes, 30 seconds

The first two lines in the generated file will set the program origin to the original starting address specified by the user, and will set the target address of the object code to \$D000. This is done to prevent the assembler's default OBJ of \$7000 from causing a large text file (such as Integer Basic) from being overwritten by assembly, and to allow modifications to be made to a program without overwriting the copy of the program which is in memory.

Once a text file has been generated and completely processed, relocation of the program to run in another section of memory usually consists of changing the program origin and assembling the file. Something to watch out for is if a program loads an address for a monitor hook (such as at \$36, \$37 or \$38, \$39) within the program by immediate loads, rather than loading the address to be placed in the hook location from a data table. If this occurs in a program and is not caught, a relocated program will not run properly. If you disassemble this program (SUPERDIS), note that the monitor hook at \$36, \$37 is set by the program with immediate loads. If this is known before the file is processed, adding a "dummy" data statement of the form:

DA \$addr,

where addr is the address placed into the monitor hook location, will cause that address to remain as a label in the file (assuming that addr lies within the program to begin with).

COMPATIBILITY WITH TED II +, V. 698:

Superdis is a co-resident with TED/EXEC and TED/EDITOR and may be BRUN from the TED EXEC module, using the D command. It is possible to pop in and out of TED with impunity as long as no attempt is made to use TED's ASM command. To ASseMble a source file that has just been SAVED by Superdis, it is necessary to RUN TED II + from disk.

Superdis, after completing the newly created source file, will process it to remove any excess spaces and automatically BSAVE it to disk, after asking the user for a name, adding the ".S" suffix required by TED.

HOW TO HANDLE A LARGE PROGRAM:

If you are disassembling a very large program (such as Integer Basic), chances are that you will not have sufficient memory space for the source file, particularly if you do not wish to overwrite the program itself, or the DOS. One way to handle this is as follows:

1. Disassemble as much of the program as you can.
2. Commence processing the file, but do not allow processing to be completed. Interrupt the processing with the RESET key. How long you let processing proceed is a matter of guesswork, aided by the table on the previous page and by experience.
3. Enter the editor, and use the Change command to replace all instances of six consecutive spaces to a single character (such as &,@, or %). Then use the Change command to replace all operands which have been converted

to labels back to absolute operands. Using Integer Basic as the example, the proper command would be

C"LE"\$E"

with all occurrences to be changed.

For a program starting elsewhere in memory, the command could be more complex, particularly for programs on the Programmer's Aid ROM, since changing all LD sequences to \$D would garble all load instructions (LDA, LDX, LDY). This must be done because the processing algorithm looks for a "\$" to signify the start of an operand. Restarting file processing without performing this step will result in the removal of all labels which have already been matched. Altering the processing algorithm to look for label matches without looking for a preceding "\$" would simplify this step, but would also allow spurious matches of labels with the arguments in HEX data statements, which would require editing of HEX lines before the file could be reassembled.

4. Once the file has been compressed, and all label operands changed back to absolute references, recommence disassembly at the address at which it was stopped, and continue until the text file is again full. Now go back to step 2, and continue the process until the complete program is in the file.

NOTE: A hazard of this method is that a late section of the program may have a backward reference to an early part of the program. If this is the first reference to that particular instruction, the label field entry for that instruction may already have been removed. You pay your money and you take your chances. This sort of thing can only be fixed by examining the processed file and manually patching in a label and changing the operand to the appropriate label.

INTEGER BASIC F4 ROM UTILITIES

This is a collection of superb Assembly Language utilities written by Steve Wozniak and Allen Baum that are found in the original F4 ROM. They are supplied on this diskette in both OBJECT and Source format. For those whose systems are capable of supporting the TED II + assembler, they may be re-located anywhere in memory convenient to the user. For Apple II + owners without Integer ROM capability, the OBJECT code may be utilized where it currently resides.

The included programs are: Mini-Assembler (a version of which is also contained in the Supplement program), Floating Point Routines, Sweet 16, PRDEC Routine, Multiply/Divide Routines and a Multiply Demo. Some of these are documented in the "Additional Reading" section on Page 11; the Multiply Demo, which was written by Dave Garson, shows the use of the Multiply/Divide Routines, and PRDEC is one of the most frequently called routines of the Integer ROM.

PRDEC will output a decimal number from 0 to 65535. To use it, Load the Accumulator with the high order byte of <number> and Load X with the low byte and CALL PRDEC. Or, store the low byte in \$F2 and the high byte in \$F3 and CALL PRDEC+4.

QUICK REFERENCE LIST - TED II+ COMMANDS AND PSEUDO-OPS

Executive Mode

#	SET DRIVE #
C	CATALOG
D	DISK COMMAND
M	MONITOR COMMAND
L	LOAD SOURCE FILE
S	SAVE SOURCE FILE
O	OBJECT CODE SAVE
A	APPEND FILES
T	TED (ENTER ED/ASM)
Q	QUIT

Edit Module

System Commands

NEW	Clear existing source file
LOmem:	Set lower limit of source
HImem:	Set high limit of source
PR#	Same as BASIC PR#
LEN	Display source length
LOAD	Load cassette source
SAVE	Save cassette source
TABS	Set or clear tab stops
List	List lines as specified
Find	Find line # or string
ASM	Assemble source file
Quit	Quit TED and enter EXEC

Entry Commands

Add	Add lines mode
Insert	Insert lines mode
Delete	Delete lines mode
COPY	Copy one range to another
Change	Change string to another
Edit	Edit line #, range or string

Edit Mode Sub-Commands

Carriage rtn

<i>or</i> Ctrl M	Accept entire line
Ctrl Q	Accept to cursor
Ctrl F	Find character
Ctrl I	Insert character
Ctrl O	Insert ctrl char
Ctrl D	Delete character
Ctrl R	Restart Edit
Ctrl C	Exit edit mode

TED System Pseudo-ops

OBJ	Set OBJ address
ORG	Set ORG address
EQU	Assign address to label
AST	Print asterisks
LST ON	Turn on listing
LST OFF	Turn off listing
PAG	Clear screen
PR #	Output to PR#
SYM	Print symbol table
END	End of program

Optional 6502 Instructions

BLT	Same as BCC
BGE	Same as BCS

Functional Pseudo-ops

ASC	Enter ASCII code
DCI	Same except last byte
HEX	Enter hex data
DW	Enter one byte of hex
DS	Reserve bytes
DA	Write label address
DB	Write label adr low

INTEGER BASIC + plus TED II +

TABLE OF CONTENTS

Introduction	2
Programs on this diskette	2
Part I The Integer Basic Plus System	3
Overview	3
Getting Started with Integer Basic Plus	4
Transferring Integer to other Diskettes	5
USER (X) Functions	5
Table of USER (X) Functions	6
DOS ONERR GOTO Function	7
Defining your own Functions	8
Stop List Feature	9
Program Compatibility	9
Part II TED II + Editor/Assembler	10
Assembly Language Whys and Wherefors	10
Additional Reading	11
Background and Features	12
Executive Mode Commands	13
The Edit Module	15
Delimiters	16
Abbreviations	16
Command Syntax	16
Edit Module Commands	17
System Commands	17
Entry Commands	18
Edit Mode Sub-Commands	20
Pseudo-ops	21
Summary of TED Pseudo-ops	21
Group A — Optional 6502 Instructions	21
Group B — TED System Pseudo-ops	21
Group C — Functional Pseudo-ops	23
Addressing Modes and Arithmetic Operators	24
Removing Excess Spaces from the Source File	25
TED Pointers and Memory Map	26
Part III Subsidiary and Utility Programs	27
Programs Relating to Integer Basic	27
Autostart ROM Supplement	27
TED Related Programs	28
Scruncher	28
Universal Disassembler	28
Superdis!	28
Features	29
Method of Use	29
General Comments	30
Compatibility with TED II +	31
How to Handle a Large Program	32
Integer Basic F4 ROM Utilities	
	Inside Back Cover