TI EXTENDED BASIC ASSEMBLY LANGUAGE CODE PROGRAMMER'S GUIDE

SECTION 1

Introduction

This document describes the use of TI9900 assembly language code from Extended Basic on the TI 99/4 Home Computer. It includes an introduction to the architecture of the 99/4, and to relevant portions of the design of Extended Basic. The minimum system configuration required to run 9900 code is a 99/4 console, a monitor, a Memory Expansion peripheral, and an Extended Basic programming language module. A cassette tape player/recorder or a floppy disk unit are required to make use of all available features.

The 9900 code is written and assembled on a host TI 990 minicomputer and downloaded to the 99/4 diskette. The file transfer utility of the Terminal Emulator II command module is useful for this procedure. If that is not available, the object code may be entered directly by using a small Extended Basic program to copy from keyboard to disk.

```
100 OPEN #1:"DSK1.PROG_A",OUTPUT,DISPLAY,FIXED 80
200 ACCEPT A$
300 IF A$="." THEN CLOSE #1 :: STOP
400 PRINT #1:A$
500 GOTO 200
```

SECTION 2

Memory Architecture of the 99/4

The 99/4 has several independent address spaces, which are accessed through memory mapped I/O in the primary (CPU) address space.

2.1 VDP RAM

Most of the RAM within the console is in the VDP (Video Display Processor) space. This memory is accessed through the VDP chip, and therefore cannot contain 9900 code or workspaces. VDP RAM is used for the screen image, character pattern tables, color tables, etc. When using Extended Basic with an Expansion RAM peripheral, the VDP RAM is also used to hold the program's symbol table, value stack, and string space. VDP RAM is accessed by writing the target address to CPU address >8C02 (least-significant byte first), and then reading data from address >8800 or writing data to address >8C00. The address written to >8C02 should have >4000 added to it if you wish to write. Access to VPD RAM is in auto-increment mode, so that you only need to send one address to read or write a block of data. VDP RAM consists of 16K bytes, with addresses >0000 to >3FFF (or >4000 to >7FFF for write mode).

The Screen is at VDP RAM locations >0 to >2FF. The first 32 bytes are the characters on the first line, the next 32 the characters on the second line, etc.

CAUTION

> The characters on the screen are not in the standard ASCII code. Each character has an offset of >60 added to it. Thus the character "A" is represented by a >A1, instead of by a >41. If you want to display a string on the screen, you must add >60 to each byte before writing it to the screen.

The character pattern table begins at VDP location >400. The locations >400 to >407 contain the bits for the character SPACE

(ASCII >20 or screen >80). These are initialized to all 0's, since a >20 is supposed to display as blank. Locations >408 to >40F describe the character "!" (ASCII >21 or screen >81), etc.

```
Pointers                                          VDP RAM
from CPU RAM                                      Addresses
------------                                      ---------
            +-------------------------------+ >3FFF
            |     Static Symbol Table       |
>832E --->  +-------------------------------+
            | Dynamic Sym Tab and PAB's     |
>831B --->  +-------------------------------+
            |         String Space          |
>831A --->  +-------------------------------+
            |           (free)              |
>836E --->  +-------------------------------+
            |         Value Stack           |
            +-------------------------------+ >0960
            |      Edit-Recall Buffer       |
            +-------------------------------+ >08C0
            |        Crunch Buffer          |
            +-------------------------------+ >0820
            |        Color Tables           |
            +-------------------------------+ >0800
            |    Sprite Velocity Block      |
            +-------------------------------+ >0780
            |      Character Tables         |
            +-------------------------------+ >0400
            |        GPL Interpreter        |
            |         Roll-out Area         |
            +-------------------------------+ >0300
            |      BASIC Temporaries        |
            +-------------------------------+ >0370
            |    Sprite Attribute List      |
            +-------------------------------+ >0300
            |            Screen             |
            +-------------------------------+ >0000
```

Figure 2.1

VDP RAM Usage by Extended Basic when Memory Expansion is Present.

2.2 Internal RAM

The only directly addressable CPU RAM inside the console is at addresses >8300 to >83FF. >8300 to >8313 will be used to support parameter passing to machine language subprograms. This will be discussed later under CALL LINK. The rest is used by Extended Basic for Pointers into VDP RAM (e.g. top-of-stack) and global parameters (e.g. OPTION BASE 0 or 1, ON ERROR transfer address, etc.).

2.3 Expansion RAM

The Memory Expansion peripheral has two blocks of memory, a 24K block from address >A000 to >FFFF, and an 8K block from >2000 to >3FFF. Extended Basic uses the 24K block for storage of the Basic program, line number table, and numeric variables. The 8K block may be used for machine language subprograms. It is possible to use a part of the 24K block for machine language as well. Basic allocates space to itself from the high end of memory, so you may place machine language code and data at the low end. In order to load code here, it should be assembled in absolute mode (e.g. AORG >A000) rather than relocatable. A pointer to the beginning of free (unallocated) memory may be found in location >8386 (RAMFRE).

2.4 Structure of the ALC block

The Routine Name Table consists of 8 byte entries; 6 bytes of subprogram name followed by 2 bytes of entry address. The first entry is placed at location >3FF8->3FFF As described below, these entries are taken from the 5- and 6- tags of an object file. The Last Free Address (LFA) pointer is initialized to >4000. It points at the first used byte in the Routine Name Table when the name table is not empty. The First Free Address (FFA) pointer is initialized to the first unused location after the utilities and points to the next free byte after subprograms have been loaded. FFA & LFA are loaded into RAM locations >8308 & >830A (CPU RAM) respectively at the start of a LOAD from a file. So if absolute code is to change the values of FFA & LFA it must modify locations >8308 & >830A.

```
          +-------------------------------+
>2000     |  XML Vector                   |
          +-------------------------------+
>2002     |  First Free Address Ptr (FFA) |
          +-------------------------------+
>2004     |  Last Free Address Ptr  (LFA) |
          +-------------------------------+
>2006     |  Constant  >AA55              |
          +-------------------------------+
>2008     |  Utility Branch Table         |
          +-------------------------------+
          |  Name Link Routine            |
          +-------------------------------+
          |  Utilites                     |
          |                               |
          +-------------------------------+
          |  ALC Routine Space            |
          |                               |
          |\/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/|
          |                               |
          |                               |
          |                               |
          |                               |
          |                               |
          |                               |
          |                               |
          |/\ /\ /\ /\ /\ /\ /\ /\ /\ /\ /\|
          |                               |
          |  Routine Name Table           |
>3FFF     |_______________________________|
```

Figure 2-2
ALC Support Memory Usage

SECTION 3

Interface with Extended Basic

Four subprograms are included in Extended Basic which interface between Basic and machine language.

3.1 CALL INIT

INIT initializes the free space and entry name table pointers, and transfers the standard assembly language support package from GROM to RAM. An exception will occur if INIT is called with no expansion RAM attached, or with the expansion RAM not powered up.

3.2 CALL PEEK(address,var-1,[var-2,...])

PEEK is used to directly read bytes of CPU RAM into Basic variables. The address is a decimal value from -32768 to 32767, representing two byte addresses. Addresses above >7FFF are written as negative numbers, treating the two byte quantity as a two's complement integer. (i.e. to access an address above 32767, subtract 65536 from it.)

CAUTION

Because of the autoincrement addressing of some memory mapped address spaces, PEEKing certain addresses will have undesirable side effects. In particular, trying to read from the ROM containing Extended Basic can cause a system crash.

3.3 CALL LOAD(object-1,[object-2,...])

The LOAD routine is used to load an assembly language object file or direct data into the RAM expansion for later execution

with the LINK routine. Each of the object arguments must be either a string expression or a list of integers. The string expression should evaluate to the name of a file containing a 9900 object program. The list of integers (poke list) should be an address between -32768 and 32767 (see PEEK), followed by a list of integers to be used as one byte of data each. These will be loaded into consecutive locations, starting at the given address. An empty string ("") may be used to separate the last byte of one poke list and the starting address of the next. The address in a poke list is absolute and the data is non-relocatable. If you load a routine directly with a poke list, you should also load a name table entry, so that the routine may be found by CALL LINK described below. When poking data, space is not automatically reserved. The poke procedure written by the user can do this by modifying the FFA and LFA pointers. However this cannot be done without reading (peeking) them. The PEEK routine can be used to read the pointers before modifying them.

The file will be opened and read by the LOAD routine. Relocatable code will be loaded at the first available address. Room will be reserved for the ALC routine according to the length specified in the "0-tag" field in the object file. Absolute code will be loaded at the absolute address specified in the object code.

CAUTION

> Absolute code will be loaded at whatever address is specified in the object code. Space will not be reserved in the RAM expansion for that code. Space will be reserved at the first available address for the length specified in the "0-tag" field. The assembly language programmer must take extreme care that absolute code is really needed and that it will work properly. Loading data into memory already being used by Extended Basic can, of course, cause a system crash.

The first ALC module loaded will load starting immediately after the utilities. The First Free Space pointer will be initialized to that address by the CALL INIT. Relocatable code will be relocated to the starting load address. Whenever a "0-tag" is encountered the starting load address is updated from the First Free Space pointer and the First Free Space pointer has the

module length added to it. With this scheme modules are loaded serially from the lowest address. Individual modules cannot be deleted from memory. The whole memory may be cleared with a CALL INIT.

The object tags generated by DEF statements (5- and 6-tags) are used to define ALC routines which can be called by name from BASIC. The 5- and 6- tags contain the DEFed name and address. These names and addresses are placed in the Routine Name Table starting at the highest memory address (>3FFF) and working down. These entries are 8 bytes each.

Assembly language code may be loaded from any device which supports sequential display files of fixed length 80. Principally the load device will be a file on the floppy disk, however, the audio cassette recorder may be used.

In order for the object code files to load properly the following conventions must be followed:

Program: No secondary references (SREF, LOAD) or segments (PSEG, DSEG, CSEG) are allowed.

Assembly: The SYMT (symbol table) option cannot be used.

Linking: Following is the correct form for the Link Editor Control File:

```
TASK  task-name
PARTIAL
NOTGLOBAL label,...
INCLUDE  file-name
INCLUDE  file-name
 "   "    "    "

END
```

If only one file is being used then it does not have to be linked.

The PARTIAL command retains the entry points (5- and 6-tags) which are used by the BASIC subprogram LINK as entry points. Libraries may be used in the link edit.

The NOTGLOBAL command causes the labels listed to be excluded from the Linked Object File. They will still be used during the Link process though. This would include any labels that were DEFD but are not to be used as entry points from Extended Basic.

If the DEFS or REFS macros are used then all of those labels will have to be included in a NOTGLOBAL command, otherwise they would be placed into the Routine Name Table (and be available as entry points from Extended Basic). A skeleton Link Edit Control File is available on file: .HCDEV.ALCLINK which contains the NOTGLOBAL commands for the labels DEFD by DEFS. Specifically file .HCDEV.ALCLINK should contain:

```
TASK
PARTIAL
NOTGLOBAL VDPWA,VDPWD,VDPRD,VDPSTA,FAC,SUBWS,GPLWS
NOTGLOBAL NUMASG,NUMREF,STRASG,STRREF,XMLLNK,KSCAN
NOTGLOBAL VSBW,VMBW,VSBR,VMBR,VWTR,ERR,FADD
NOTGLOBAL FSUB,FMUL,FDIV,SADD,SSUB,SMUL,SDIV,CSN
NOTGLOBAL CFI,FCOMP,NEXT,COMPCT,GETSTR,MEMCHK,VPUSH
NOTGLOBAL ASSGNV,CNS,VPOP,CIF,SCROLL,VGWITE,GVWITE
END
```

3.4 CALL LINK(string-expression,[parameter-list])

The LINK subprogram passes control from a BASIC program to an assembly language subprogram. The string-expression should evaluate to 1-6 characters, which is the name of an assembly language routine. The parameter-list is an optional list of parameters, passed using the same conventions as parameters passed to an Extended Basic subprogram. The LINK subprogram performs the following actions:

1. Evaluate the ALC subprogram name and its length (1-6), and push the FAC entry on the value stack.

2. Build the ALC argument list consisting of stack and identifier table entries.

3. Move the sub-name from string space to the FAC entry and transfer control over to the utility code.

4. On return, branch to an error routine if an error has been detected; otherwise, clear the stack and return normally.

The arguments are passed to the ALC subprogram through the stack in VDP RAM and the identifier list in CPU RAM. The identifier list consists of the following;

address

1. (>8300 - >830F) Argument identifiers (one byte each; maximum of 16)

2. (>8310) Old value stack pointer.

3. (>8312) Number of arguments.

4. (>8314) Temporary storage space for subroutine name.

There are 6 types of arguments supported.

1. Numeric Expression - The identifier will contain a 0. The eight byte stack entry will contain the value of the numeric expression. The first byte is the exponent, in excess-64, radix 100 notation. The other seven bytes contain 0 to 99, for radix 100 digits. If the number is negative, the first word (two bytes) IS negated.

2. String Expression - The identifier will contain a 1. The string entry will contain:

   a. Bytes 0-1: >001C

   b. Byte 2: >65 (the string tag)

   c. Bytes 4-5: Pointer to the value of the string in VPD RAM.

   d. Bytes 6-7: Length of the string. Byte 6 should always be zero, since the maximum string length is 255 characters.

3. Numeric variable - This item will either be a numeric variable or a numeric array element. The identifier list will contain a 2 for this entry. The string will contain:

   a. Bytes 0-1: Pointer to the varaible's symbol table entry in VDP RAm.

   b. Byte 2: zero

   c. Bytes 4-5: Pointer to the eight byte value of the variable, in Expansion RAM.

4. String Variable - This item will either be a string variable or a string array element. The identifier list will contain a 3. The stack entry will contain:

a. Bytes 0-1: Pointer to the variable's symbol table entry in VDP RAM.

b. Byte 2: >65

c. Bytes 4-5: Pointer to the string's value in VDP RAM.

d. Bytes 6-7: String length.

5. Numeric Array - This entry results from an argument of the form A() or A(,) etc. This is used so that a subprogram may manipulate an entire array. The identifier list will contain a 4 for this entry. The string entry will contain:

   a. Bytes 0-1: Pointer to the array's symbol table entry in VDP RAM. The byte pointed to will contain the number of dimensions of the array in the least significant three bits.

   b. Byte 2: zero.

   c. Bytes 4-5: Pointer to the array's value space in VDP RAM. The value space will have two bytes for each dimension, indicating the maximum index for that dimension, and two bytes for a pointer to the first element's eight byte value in Expansion RAM. The values are stored in row-major order. (i.e. the first index varies the fastest.)

6. String Array - Similar to the entry for a numeric array except the identifier list will contain a 5 and byte 2 of the stack entry will contain >65. The value space for a string array will contain two bytes for each dimension, indicating the maximum index, followed by two bytes for each array element, which is used as a pointer to the element's value string in VDP RAM.

Any argument that can be passed as a variable will be passed as such rather than as an expression.

The assembly subprogram is called through a Name Link Routine written in assembly language and loaded with the utilities. The Name Link Routine looks up the name of the routine to be called in the Routine Name Table and branches to the ALC subprogram. The Name Table is searched from the lowest address up so that if two routines are LOADed with the same name then the second one loaded will be used. The name of the routine to be called will be placed in FAC blank filled to 6 characters.

If the name supplied by the user is greater than 6 characters then an error will result. This error will be detected in GPL before the XML. If the Name Link Routine cannot find the name in the table then it will return with the GPL condition bit set to indicate that error.

The Name Link Routine will branch to the ALC routine with a direct 9900 code branch. When the ALC routine is called from the LINK routine the workspace will be at >83E0. The ALC routine must retain R11, R13, R14, and R15 in that workspace and return to the GPL interpreter with an RT instruction. Normally the ALC routines will have their own workspaces and will return to GPL with a LWPI >83E0 followed by a RT. The macros ENTRY and RETURN described below are useful to handle this.

The ALC subprogram can assign new values to numeric or string variables or to elements of numeric or string arrays with utilities provided by the system. These utilities are described in a later section.

When the ALC routine returns to LINK the stack must be cleared. The entries on the stack must be individually popped to release any temporary strings.

SECTION 4

Utilities

The utility routines are provided for use by assembly language subprograms to access machine resources and to interface with the BASIC interpreter.

Utility subroutines are provided in the RAM expansion so that ALC subprograms can be shorter and more machine independent. The utility routines are called with BLWP instructions and use registers to pass arguments.

4.1 VDP Access Utilities

Several utilities are provided for access to the Video Display Processor. Macros (see below) are also provided when speed is important. Generally using the utilities instead of the macros will save code space.

All parameters are passed through the calling program's workspace registers.

<u>VDP Single Byte Write</u>. This routine writes the single byte value in the most significant byte of register 1 to the VDP RAM address indicated in register 0. The routine is accessed by a BLWP @VSBW.

<u>VDP Multiple Byte Write</u>. This routine writes the number of bytes indicated in register 2 from the CPU RAM buffer pointed to by register 1 to the VDP RAM buffer pointed to by register 0. The utility is called with a BLWP @VMBW.

<u>VDP Single Byte Read</u>. This routine reads a single byte from the VDP RAM address indicated in register 0, and places it in the most significant byte of register 1. The routine is called with a BLWP @VSBR.

<u>VDP Multiple Byte Read</u>. This routine reads the number of bytes indicated in register 2 from the VDP RAM buffer pointed to by register 0, and places them in the CPU RAM buffer pointed to by register 1. The utility is accessed by a BLWP @VMBR.

VDP Write to Register. This routine writes the single byte value in the least significant byte of register 0 to the VDP register indicated in the most significant byte of register 0. The routine is called with a BLWP @VWTR.

4.2 Argument passing utilities

Numeric Assignment. A utility is provided to allow a value to be assigned to a numeric variable passed as an argument. The floating point variable to be assigned will be in FAC and the argument number will be passed in R1 (full word). The utility is called with a BLWP @NUMASG. If the requested argument is not a numeric variable then the routine will return to GPL with the GPL condition bit set.

For assignments to a simple numeric variable R0 must contain a zero. For an assignment to an array R0 will contain the array element number. The assignment utility will test for legal bounds on the element number. With OPTION BASE 0 the element number must range from 0 to (maximum number of elements - 1). With OPTION BASE 1 then the element number must range from 1 to maximum number of elements.

String Assignment. A utility is provided to allow a string to be assigned to a string variable passed as an argument to the subprogram. The utility does the following:

1. Allocates space for the string in VDP RAM.
2. Copies the string into VDP RAM.
3. Assigns the string to the selected varaible.
4. Fixes up the original argument stack entry to point to the new string.

The string to be assigned is constructed by the ALC routine in the RAM expansion. The first byte of the string is the length of the string. The assignment utility is called with the string address in register 2 and the argument number in register 1 (full word). The utility is called with a BLWP @STRASG. If the argument specified is not a string variable then the routine will return to GPL with the GPL condition bit set.

For assignments to a simple string variable R0 must contain a zero. For an assignment to an array R0 will contain the array element number. The assignment utility will test for legal

bounds on the element number. With OPTION BASE 0 the element number must range from 0 to (maximum number of elements - 1). With OPTION BASE 1 then the element number must range from 1 to maximum number of elements.

Get Numeric Parameter. A utility is provided to fetch the value of a numeric parameter. Register 1 contains the parameter number. If the parameter is an array, register 0 contains the element number, otherwise, register 0 contains 0. The value of the numeric parameter is returned in FAC. The utility routine is accessed by a BLWP @NUMREF.

Get String Parameter. This utility is used to fetch the value of a string parameter. Register 1 contains the parameter number. If the parameter is an array, register 0 contains the element number, otherwise, register 0 contains 0. Register 2 contains the address of a CPU RAM buffer. Upon calling the utility, the first byte of the buffer must contain the buffer length. If the string will not fit in the buffer, an error condition occurs. Otherwise, the string is returned in the buffer following the length byte. The length byte will be modified to reflect the actual length of the string. This utility is called with a BLWP @STRREF.

4.3 Extended Utilities

Utilities built into the ROM of the 99/4 console and the ROMs in Extended BASIC may be accessed as follows:

```
BLWP @XMLLNK
DATA <routine-name>
```

Some routine names will BL through the XML tables in Extended BASIC While others will call various console ROM routines. Routine names are defined in the EQUS macro and are li

ed below.

FADD	Floating point add.
FSUB	Floating point subtract.
FMUL	Floating point multiply.
FDIV	Floating point divide.
SADD	Floating point stack add.
SSUB	Floating point stack subtract.

SMUL Floating point stack multiply.

SDIV Floating point stack divide.

CSN Convert string to number.

CFI Convert floating point to integer.

FCOMP Floating point compare.

COMPCT Perform garbage collection.

GETSTR Allocate space from string space.

MEMCHK Allocate memory for PAB.

VPUSH Push a value on the floating point stack.

ASSGNV Assign value to variable.

CNS Convert number to ASCII string.

VPOP Pop a value from the stack.

CIF Convert integer to floating point.

SCROLL Scroll the screen.

VGWITE Transfer VDP memory to expansion RAM.

GVWITE Transfer expansion RAM to VDP memory.

<u>Keyboard Scan</u>. The equivalent of a GPL scan instruction can be done by a BLWP @KSCAN. The GPL status bit may be tested on return with a compare ones corresponding (COC) instruction. The GPL status register is in location >83C7.

	/ H /	GT /	COND /	CARRY/	OVF /	0 /	0 /	0 /
bit	7	6	5	4	3	2	1	0

Bit 5 will be set if a key was found pressed, and it was different from the key found pressed on the last call to KSCAN. The program must select keyboard device by placing a byte in location >8374. The meaning of this byte is the same as the key-unit in the CALL KEY subprogram. You may read the ASCII value of the key pressed from location >8375. Joystick Y and X positions are in locations >8376 and >8377, respectively.

Error Reporting. The assembly language program may report any existing Extended BASIC error or warning message upon returning to BASIC. This is done with the error reporting utility. The program must load R0 with the appropriate error code and then BLWP @ERR. The definitions of the error codes may be included in the standard macro file, described in a then next section.

SECTION 5

Macros

Assembler macros are provided to make programming in the TI 99/4 environment easier. The macros are used in a 9900 assembly with a COPY of the file given in Appendix A.

5.1 EQUS

This EQUS macro is used to define global equates for the ALC subprogram. Included is the definition of a workspace and equates for the various hardware resources. Equates for the addresses of the utility branch table entries are also provided.

5.2 EQUDEB

This macro is similar to EQUS, but contains the addresses appropriate for use from a debugger station. The addresses of routines in console ROM are different, since extra code has been included there to support breakpoints.

5.3 ERREQU

This macro provides equates for all the various Extended BASIC error and warning messages.

5.4 DEFS

The DEFS macro is used when several ALC routines will be linked together to form one ALC subprogram. This macro causes the definitions in the EQUS macro to be global during the link edit.

5.5 REFS

The REFS macro is used when several ALC routines will be linked together to form one ALC subprogram. This macro allows a routine to reference the equates defined in the DEFS macro.

5.6 ENTRY

The ENTRY macro is written as ENTRY label where label is the entry point name of the routine. This macro defines the entry point with the specified, defines a workspace for the subprogram, and loads that workspace.

5.7 RETURN

The RETURN macro is used to return from the ALC subprogram to BASIC. This macro requires that the program contain the one byte value >20 at the label "C20". This value is used to reset the GPL condition bit before returning to BASIC. If this is not done, false error reporting can occur.

5.8 VDPADR

This macro loads the VDP address with the specified workspace register.

5.9 VDPWD

The VDPWD macro is used to write one byte of data from the most significant byte of a register. Two VDPWD macros cannot be used without other code between them or the accesses will be too fast for the VDP.

5.10 VDPRD

The VDPRD macro is used to read one byte of data from the VDP into the most significant byte of a register. Two VDPRD macros cannot be used without other code between them or the accesses will be too fast for the VDP.

5.11 CALL Extended Utilities

A macro is provided to call utilities built into the ROM of the 99/4 console and the ROMs in Extended BASIC. The syntax and code generated are:

```
CALL <routine-name>

BLWP @XMLLNK
DATA <routine-name>
```

5.12 SCAN

The SCAN macro scans the 99/4 keyboard. The protocol is identical to the GPL SCAN instruction. The GPL status bit may be tested on return with a compare ones corresponding (COC) instruction.

5.13 SOUND

The SOUND macro can be used to set up for the execution of auto-sound. The sound list, as described in the "GPL Programmer's Guide", must be placed somewhere in VDP RAM. The VDP RAM address must be on a word boundary, ie. an even value, and must be in the register used in the macro. In addition the use of this macro requires that the program contain the one byte value >01 at the label "C01". This code does not actually start the auto-sound. It simply initializes all the necessary memory locations. Auto-sound is driven by the VDP interrupt, therefore, the sound list will not begin executing until the interrupt is turned on.

5.14 ERROR

The ERROR may be used to report errors and warnings. If it becomes necessary for an ALC program to return to BASIC with an error condition, the program may use the following macro in place of the normal return.

```
ERROR <symbol or value>
```

SECTION 6

Development

The assembly language support will be developed using the GPL debugger hardware. This causes some difficulties since the debugger preempts RAM expansion at address >2000->3FFF. That space is used by code and RAM memory for debug purposes. To avoid this problem the ALC support will be developed using RAM at >E000->FFFF. To achieve this Extended BASIC will be patched (in GROM) to only use 16K of the 24K RAM block. Then BASIC will use >A000->DFFF for program and data storage and >E000->FFFF for ALC support. Reassembly for EGROM will change the appropriate equates to use the correct address. The debugger EPROMS will be changed so that the XML table at >2000 will be moved to >E000.

Breakpoint capability for assembly language is being added to the debugger to support testing the ALC support and for use during ALC development.

Any ALC code developed on the debugger will run at a different address than in the real machine due the reasons mentioned above. For relocatable code (the recommended way) this should be transparent when the code is written.

SECTION 7

Notes

7.1 PEEK and POKE

The LOAD statement supports the equivalent of a POKE function (direct memory load). The values PEEKed or POKEed are one byte values in the range 0 to 255. Larger values are put in this range (i.e. 256=0, 257=1 ...), provided they do not exceed the range 32767 to -32768. If the address for PEEK or POKE, or the data for POKE is a floating point value than it is rounded.

SECTION 8

Errors

No new error messages were added for INIT, LOAD, or LINK. All of the error messages issued are from Extended Basic. This section lists the causes of errors for INIT, LOAD, and LINK that are unique to assembly language support.

8.1 INIT

1. Syntax Error

   a. Expansion Ram not present

8.2 LOAD

1. Syntax Error

   a. INIT has not been called

2. Numeric Overflow

   a. Address of POKE out of range +32767 to -32768

   b. Data not in range
      32767 to -32768

3. Unrecognized Character

   a. Invalid tag field

4. Data Error

   a. Check Sum error when reading file

5. Memory Full

   a. Not enough memory to load relocatable code into

32767 to -32768.

8.5 NUMASG, STRASG, NUMREF, STRREF

1. String Number Mismatch

   a. Requested argument is not the proper type (string vs. numeric).

2. Bad Argument

   a. Attempt to do an assignment to an expression instead of a variable.

   b. Array element number (R0) is non-zero value when doing an assignment or reference with a simple, i.e. non-array, argument.

   c. Argument number (R1) is zero.

   d. Argument number (R1) exceeds number of arguments.

3. Bad Subscript

   a. Array element number (R0) is zero when OPTION BASE is 1.

   b. Array element number (R0) exceeds number of array elements.

   c. Bad dimension information found in symbol table.

4. String Truncated

   a. In using STRREF, the referenced string exceeds the length of the provided buffer.

APPENDIX A

Macro Definition File

This appendix lists the text of the file that should be included in an assembly language program (using the COPY directive) to define the macros discussed in an earlier section.

```
*
*       MACROS FOR ALC SUPPORT
*
EQUDEB $MACRO
ON      EQU  1
OFF     EQU  0
DEBUG   EQU  ON
VDPWA   EQU  >8C02
VDPWD   EQU  >8C00
VDPRD   EQU  >8800
VDPSTA  EQU  >8802
FAC     EQU  >834A
SUBWS   BSS  32
GPLWS   EQU  >83E0
*
*       utility branches
*
NUMASG  EQU  >E008
NUMREF  EQU  >E00C
STRASG  EQU  >E010
STRREF  EQU  >E014
XMLLNK  EQU  >E018
KSCAN   EQU  >E01C
VSBW    EQU  >E020
VMBW    EQU  >E024
VSBR    EQU  >E028
VMBR    EQU  >E02C
VWTR    EQU  >E030
ERR     EQU  >E034
FADD    EQU  >0DE4
FSUB    EQU  >0DE0
FMUL    EQU  >0EEC
FDIV    EQU  >1058
SADD    EQU  >0DE8
SSUB    EQU  >0DD8
SMUL    EQU  >0EF0
SDIV    EQU  >105C
CSN     EQU  >1212
CFI     EQU  >131C
FCOMP   EQU  >0D9E
NEXT    EQU  >008E
COMPCT  EQU  >00
GETSTR  EQU  >02
MEMCHK  EQU  >04
CNS     EQU  >06
VPUSH   EQU  >0E
VPOP    EQU  >10
ASSGNV  EQU  >18
CIF     EQU  >20
SCROLL  EQU  >26
```

```
VGWITE EQU  >34
GVWITE EQU  >36
       $END
*
EQUS   $MACRO
ON     EQU  1
OFF    EQU  0
DEBUG  EQU  OFF
VDPWA  EQU  >8C02
VDPWD  EQU  >8C00
VDPRD  EQU  >8800
VDPSTA EQU  >8802
FAC    EQU  >834A
SUBWS  BSS  32
GPLWS  EQU  >83E0
*
*      utility branches
*
NUMASG EQU  >2008
NUMREF EQU  >200C
STRASG EQU  >2010
STRREF EQU  >2014
XMLLNK EQU  >2018
KSCAN  EQU  >201C
VSBW   EQU  >2020
VMBW   EQU  >2024
VSBR   EQU  >2028
VMBR   EQU  >202C
VWTR   EQU  >2030
ERR    EQU  >2034
FADD   EQU  >0D80
FSUB   EQU  >0D7C
FMUL   EQU  >0E88
FDIV   EQU  >0FF4
SADD   EQU  >0D84
SSUB   EQU  >0D74
SMUL   EQU  >0E8C
SDIV   EQU  >0FF8
CSN    EQU  >11AE
CFI    EQU  >12B8
FCOMP  EQU  >0D3A
NEXT   EQU  >0070
COMPCT EQU  >00
GETSTR EQU  >02
MEMCHK EQU  >04
CNS    EQU  >06
VPUSH  EQU  >0E
VPOP   EQU  >10
ASSGNV EQU  >18
CIF    EQU  >20
```

```
SCROLL EQU  >26
VGWITE EQU  >34
GVWITE EQU  >36
       $END
ERREQU $MACRO
*
*      ERROR EQUATES
*
ERRNO  EQU  >0200        2   Numeric Overflow
ERRSYN EQU  >0300        3   Syntax Error
ERRIBS EQU  >0400        4   Illegal after subprogram
ERRNQS EQU  >0500        5   Unmatched quotes
ERRNTL EQU  >0600        6   Name too long
ERRSNM EQU  >0700        7   String-num mismatch
ERROBE EQU  >0800        8   Option base error
ERRMUV EQU  >0900        9   Improperly used name
ERRIM  EQU  >0A00       10   Image error
ERRMEM EQU  >0B00       11   Memory full
ERRSO  EQU  >0C00       12   Stack overflow
ERRNWF EQU  >0D00       13   Next without for
ERRFNN EQU  >0E00       14   For next nesting
ERRSNS EQU  >0F00       15   Must be in subprogram
ERRRSC EQU  >1000       16   Recursive subprogram call
ERRMS  EQU  >1100       17   Missing subend
ERRRWG EQU  >1200       18   Return without gosub
ERRST  EQU  >1300       19   String truncated
ERRBS  EQU  >1400       20   Bad subscript
ERRSSL EQU  >1500       21   Speech string too long
ERRLNF EQU  >1600       22   Line not found
ERRBLN EQU  >1700       23   Bad line number
ERRLTL EQU  >1800       24   Line too long
ERRCC  EQU  >1900       25   Can t continue
ERRCIP EQU  >1A00       26   Command illegal in program
ERROLP EQU  >1B00       27   Only legal in a program
ERRBA  EQU  >1C00       28   Bad argument
ERRNPP EQU  >1D00       29   No program present
ERRBV  EQU  >1E00       30   Bad value
ERRIAL EQU  >1F00       31   Incorrect argument list
ERRINP EQU  >2000       32   Input error
ERRDAT EQU  >2100       33   Data error
ERRFE  EQU  >2200       34   File error
ERRIO  EQU  >2400       36   I/O error
ERRSNF EQU  >2500       37   Subprogram not found
ERRPV  EQU  >2700       39   Protection violation
ERRIVN EQU  >2800       40   Unrecognized character
WRNNO  EQU  >2900       41   Numeric overflow
WRNST  EQU  >2A00       42   String truncated
WRNNPP EQU  >2B00       43   No program present
WRNINP EQU  >2C00       44   Input error
WRNIO  EQU  >2D00       45   I/O error
```

```
WRNLNF EQU  >2E00          46   Line not found
       $END
*
DEFS   $MACRO
       DEF  VDPWA,VDPWD,VDPRD,VDPSTA,FAC,SUBWS,GPLWS
       DEF  NUMASG,NUMREF,STRASG,STRREF,XMLLNK,KSCAN
       DEF  VSBW,VMBW,VSBR,VMBR,VWTR,ERR,FADD
       DEF  FSUB,FMUL,FDIV,SADD,SSUB,SMUL,SDIV,CSN
       DEF  CFI,FCOMP,NEXT,COMPCT,GETSTR,MEMCHK,VPUSH
       DEF  ASSGNV,CNS,VPOP,CIF,SCROLL,VGWITE,GVWITE
       $END
*
REFS   $MACRO
       REF  VDPWA,VDPWD,VDPRD,VDPSTA,FAC,SUBWS,GPLWS
       REF  NUMASG,NUMREF,STRASG,STRREF,XMLLNK,KSCAN
       REF  VSBW,VMBW,VSBR,VMBR,VWTR,ERR,FADD
       REF  FSUB,FMUL,FDIV,SADD,SSUB,SMUL,SDIV,CSN
       REF  CFI,FCOMP,NEXT,COMPCT,GETSTR,MEMCHK,VPUSH
       REF  ASSGNV,CNS,VPOP,CIF,SCROLL,VGWITE,GVWITE
       $END
*
ENTRY  $MACRO P1
       DEF :P1
:P1:   LWPI SUBWS
       $END
*
RETURN $MACRO
       SZCB @C20,@>837C
       LWPI GPLWS
       B    @NEXT
       $END
*
VDPADR $MACRO P1
       SWPB :P1
       MOVB :P1:,@VDPWA
       SWPB :P1
       MOVB :P1:,@VDPWA
       $END
*
VDPWD  $MACRO P1
       MOVB :P1:,@VDPWD
       $END
*
VDPRD  $MACRO P1
       MOVB @VDPRD,:P1
       $END
*
CALL   $MACRO P1
       BLWP @XMLLNK
       DATA :P1
```

```
        $END
*
SCAN    $MACRO
        BLWP @KSCAN
        $END
*
SOUND   $MACRO P1
        MOV  :P1:,@>83CC
        SOCB @C01,@>83FD
        MOVB @C01,@>83CE
        $END
*
ERROR   $MACRO P1
        LI   R0,:P1
        BLWP @ERR
        $END
```