# Software Considerations for the VIC64

## Introduction

This application note provides the VIC64 software developer with proven tips and examples for both configuring and operating the VIC64. The software described here is based on a SPARC-based VMEbus card utilizing a VIC64. This board was developed within Cypress Semiconductor as a test/evaluation vehicle for the VIC64 and the CY7C964.

This application note also discusses the configuration of the CY7C964 VMEbus address compare functions.

Although this application note specifically addresses the VIC64, virtually everything in this application note could also be applied to the VIC068A. VIC64-only features are flagged to notify the reader of items that are not applicable to the VIC068A.

The source files vic.h, eval_bd.h, and blt_cmd.c, which are described in this application note, are available through the Cypress Semiconductor BBS (Bulletin Board System). These files are contained within a file named “SAMPCODE.EXE.”

### Related Documents

The reader may also wish to consult the following documents for additional information:

- *VIC068A/VAC068A User’s Guide*
- *VIC64/CY7C964 Design Notes*

These documents are available through your local Cypress Semiconductor field sales office.

## Hardware Overview

The examples in this application note are based on an actual design of a SPARC-based VIC64 evaluation VMEbus board developed by Cypress Semiconductor. The following paragraphs provide background for this hardware platform. Contact your local field applications engineer regarding specific hardware information on this board.

### The Evaluation Board

This evaluation board includes the following features:

- Cypress’s CY7C611 embedded SPARC microprocessor
- Floating-point support
- 64 Kbytes to 4 Mbytes of private SRAM
- 64 Kbytes to 2 Mbytes of shared SRAM
- 512 Kbytes of EPROM for the embedded monitor program
- Performs D64 VMEbus transfers utilizing VIC64 and CY7C964 devices
- MC68681 DUART
- 2 Kbytes of non-volatile storage
- Real-time clock

### Evaluation Board Local Control Register (LCR)

The evaluation board contains a single 32-bit, dual-purpose control register. When read, this register provides the memory size of the SIMM sockets as shown in *Table 1*.

**Table 1. LCR Read Fields**

| Bits | Socket |
|---|---|
| bits 0,1 | SIMM socket 1 size (private) |
| bits 2,3 | SIMM socket 2 size (private) |
| bits 4,5 | SIMM socket 3 size (private) |
| bits 6,7 | SIMM socket 4 size (private) |
| bits 8,9 | SIMM socket 5 size (shared) |
| bits 10,11 | SIMM socket 6 size (shared) |

The two bits for each SIMM contain one of the codes shown in *Table 2*.

**Table 2. SIMM Size Codes**

| Code | Size |
|---|---|
| 00 | 1M-byte SIMM |
| 01 | 256K-byte SIMM |
| 10 | 64K-bytes SIMM |
| 11 | Socket empty |

When written, this register provides control over the resources shown in *Table 3.*

**Table 3. LCR Write Fields**

| Bits | Function |
|---|---|
| bits 0–15: | LEDs (lit when bit is clear) |
| bits 16,17: | VIC64 reset |
| bits 18–28: | VMEbus address (A31:21) |
| bits 29,30: | VMEbus address size (ASIZ0-1) |
| bit 31: | VMEbus data port size (WORD*) |

Bits 0–15 provide control of 16 LEDs located on the edge of the board. When a bit is cleared, the corresponding LED is lit.

Bits 16 and 17 provide control over the reset operations of the VIC64. When bit 16 is cleared, the board’s state logic asserts the IRESET* signal of the VIC64. When bit 17 is cleared, state logic asserts the IPL0* signal of the VIC64, issuing a global reset to the VIC64.

Bits 18-28 provide control over the most-significant eleven VMEbus address lines. Prior to a VMEbus access, this bit field is loaded with the most-significant eleven address bits. An access is then made to a predefined address (VME_BASE_ADRS) with the least-significant 21 VMEbus

address lines obtained from the physical address of the transaction.

Bits 29 and 30 control the VIC64 ASIZ0/1 signals respectively. These signals tell the VIC64 what address size to use.

Bit 31 controls the WORD* signal line. When clear, the VIC64 performs D16 VMEbus accesses; when set, D32.

## Software Considerations

***SPARCmon***™

The embedded monitor program used on the evaluation board is *SPARCmon*. *SPARCmon* is a commercial product available from Sun Microsystems. *SPARCmon* consists of source code modules for initialization, trap handling, floating-point support, process control, remote debugging, I/O, and a main command interpreter. Board-specific code such as board initialization, test, and additional commands are incorporated into *SPARCmon* separately. This application note does not address the specifics of *SPARCmon*, only board-specific details as it relates to the VIC64.

**Boot-Up**

The flow of initialization for booting the evaluation board is described in the following sections.

*Disable Traps*

Traps are disabled until resources exist to service them.

*Initialize 7C611 Window Invalid Mask (WIM) and Trap Base Register (TBR)*

*Reset the VIC64*

This is discussed in detail later in this application note.

*Test First 64 Kbytes of Private Memory*

This provides us with tested memory for temporary storage to perform subsequent boot tasks.

*Set Up Initial Stack Frame Pointer and Enable Traps*

With the first 64 Kbytes of memory tested, we may now service traps. The trap vector table is located initially in EPROM at address $0.

*Initialize I/O*

This consists of setting up I/O tables, structures and the DUART itself.

*Perform Board Diagnostics*

The remainder of the board is checked, including

- — EPROM checksum
- — NVRAM checksum
- — Determining amount of SRAM installed
- — Testing remaining private SRAM
- — Testing the shared SRAM
- — Testing the NVRAM
- — Testing the VIC64 (discussed later)
- — Testing the DUART
- — Configuring board local memory map

Local memory map is created with regions for

- — Monitor variables (DATA)
- — Uninitialized monitor variables (BSS)
- — The relocated trap table
- — User memory area
- — User stack (STACK) area

*Clear User Memory Areas*

The user areas are “cleared” to a predefined value.

*Relocate the Trap Table in EPROM to SRAM*

This speeds up trap table accesses and makes the table modifiable. The TBR is adjusted after the table is moved.

*Configure VIC64*

This is discussed in detail later in this application note.

## VIC64 Initialization and Test

**VIC64 Register Accesses**

All of the VIC64’s internal registers are 8 bits wide but occupy 32 bits of address space. Specific address and size information must be presented to the VIC64 in order for the VIC64 to accept the register access.

When the VIC64 has been selected for a register access (CS*, PAS*, and DS* are asserted to the VIC64), the VIC64 checks the SIZ1/0 and LA[1:0] signals to insure proper byte orientation. This is because the VIC64 is only connected to the lower 8 data lines of the local data bus and the data must be aligned as such.

*Table 4* shows the valid combinations of SIZ1/0 and LA[1:0] that must be present for the VIC64 to accept the register access. The VIC64 mimics the Motorola CISC processors in that the SIZ and LA combinations for it are the same as for the VIC64. The SIZ codes for the CY7C611 are not the same and translation circuitry is required.

**Table 4. VIC64/068 D(7:0) Data Alignment**

| SIZ1 | SIZ0 | LA1 | LA0 | Size |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | Byte |
| 1 | 0 | 1 | 0 | Word |
| 0 | 0 | 0 | 0 | Longword |
| 1 | 1 | 1 | 1 | 3-Byte |

If *Table 4* is not satisfied, the VIC64 ignores the attempted cycle by not reading or writing the information and not acknowledging the cycle (does not assert DSACKi*).

**VIC64 Reset**

The evaluation board issues a power-on reset to the VIC64 via the LCR. The LCR contains two bits for VIC64 reset. Bit 16 controls the assertion of IRESET* for the purposes of performing a internal reset. Bit 17 controls the assertion of IPL0*, which is used in conjunction with IRESET*, to perform a global reset. The VIC64 requires that a global reset be issued at power-up. The SPARC assembler code in *Figure 1* performs a VIC64 global reset.

This routine is written in assembler language because it must be a “leaf” routine. That is, it must not use the stack in any way since no stack exists yet. Calls from a high-level language or calling an additional routine would almost certainly use the stack.

```
#include <eval_bd.h>                ; Needed for LCR pointer

set LOCAL_CONTROL_BASE_ADRS, %l6    ; This symbol points to the LCR
set 0xffffffff, %l2
st %l2, [%l6]                       ; ”clear” LCR
set 0xfff7ffff, %l2
st %l2, [%l6]                       ; Assert IRESET*
set 0xfffcffff, %l2
st %l2, [%l6]                       ; Assert IPL0*
set 0xfff7ffff, %l2
st %l2, [%l6]                       ; Remove IPL0*
set 0xffffffff, %l2
st %l2, [%l6]                       ; Remove IRESET*
```

**Figure 1. VIC64 Reset**

Notice that the VIC64 is reset in stages. First the IRESET* signal is asserted to the VIC64 by clearing bit 16 of the LCR. The next instruction clears bit 17 to assert IPL0*. The reason that these are performed in separate instructions is that sufficient time must be allowed for the assertion of IRESET* to switch the IPL0* from an output to an input. Next, the IPL0* signal is removed, then the IRESET* signal is removed, in separate instructions. This is done to insure that the VIC64 200-ms reset timeout is observed. If they were removed simultaneously, this timeout may not be observed and the reset would complete immediately. Refer to section 12.1 of the *VIC068/VAC068 User's Guide* for more details on VIC reset.

**VIC64 Test**

To determine if the VIC64 is present and has been reset properly, the VIC64 test routine performs write-read-verify cycles to the VIC64 ICR0-5 registers. At this time, the VIC64 version register is read to determine the mask revision. The mask register reads $00, and any VIC64 values above $F0 indicate a VIC068 is installed. This may or may not be acceptable for specific applications.

**VIC64 Configuration**

The configuration of the VIC64 is accomplished by writing the VIC64 registers to desired values. The board stores these predetermined values as a structure located in the NVRAM at boot-up. The VIC64 configuration routine reads these values and stores them into the appropriate VIC64 registers. This way, the configuration of the VIC64 is not hard-coded and may be modified by simply changing the values in NVRAM and calling a VIC64 configuration routine.

**VIC64 Address Spaces**

In VMEbus systems, each VMEbus board typically has its own unique address spaces within the total 4-Gbyte VMEbus addressing range. These regions may consist of various sub-regions including:

- A32, A24, and/or A16 regions
- D32 and/or D16 regions
- Interprocessor communication regions

In addition to the VMEbus address spaces, the local processor within each board works with a local address space that may include:

- Private memory
- Shared memory (shared with the VMEbus)
- UARTs
- Interrupt acknowledge
- Board control registers
- The VMEbus
- Control registers (including VIC64)

These local areas may or may not be visible to other VMEbus modules. It is not uncommon for shared memory to be the only local resource available to other VMEbus modules. This is the case for this board. The local addresses and the VMEbus addresses to this shared memory would almost certainly be different. Some type of secondary decode or address translation is necessary in these instances. In the examples given in this application note, the header file eval_bd.h defines the local address map used for the board.

The VIC64 does not directly support VMEbus accesses to their internal registers with the exception of the Interprocessor Communication registers. It is possible via external hardware to make all VIC64 registers visible to the VMEbus (see Cypress's application note titled “Using the VIC068A Without a Processor”). If a VAC068A is used, the local VIC64 (the VAC068A is not compatible with the D64 operations of the VIC64, but can be used if D64 operation are not performed) register region is fixed at addresses FFFCxxxx to FFFDxxxx. As a minimum, sufficient space must always be allotted for the 58 longwords of VIC64 registers.

**VMEbus addressing through the LCR**

As noted earlier, bits 18–28 of the LCR provide control over the most significant eleven VMEbus address lines. Therefore, a VMEbus access may consist of two parts: loading the LCR with the proper value, and performing the actual transfer to the VMEbus address location. This location consists of a fixed address in combination with the lower 21 bits of the VMEbus address.

As an example, assume a VMEbus A32, D32 read access is desired from the VMEbus address 0x38004000 and that the LEDs should remain clear (see *Figure 2*). A value of 0xA703FFFF should be written into the LCR. If the VMEbus address space on the local address map is 0xE00000 (VME_BASE_ADRS), the local address should be 0xE04000 (0xE00000 + least significant 21 bits of VMEbus address).

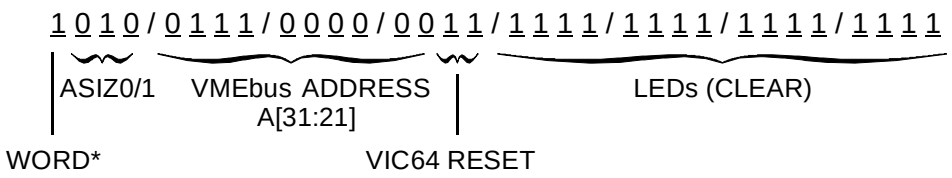


**Figure 2. VMEbus A32, D32 Read Access**

An addressing scheme of this sort makes the entire 4-Gbyte range of the VMEbus addressable by the board. A disadvantage is that the LCR must be written for any VMEbus transaction is in a different 2-Mbyte address spaces from the previous VMEbus transaction. An example of a function that would return the proper address is shown in *Figure 3*.

An example function that returns the proper LCR value could be as shown in *Figure 4*.

**CY7C964 Address Comparator Configuration**

The evaluation board uses the CY7C964 as the VMEbus slave address comparator. The address comparator consists of two registers: the mask register and the compare register. The compare register is loaded with the base address of the slave address. The mask register is loaded with a value that determines which bits of the address should be compared with the value in the compare register. This defines the size of the address region. A zero in a bit enables the comparison of the corresponding bit in the compare register to the VMEbus address bit.

For example, if there are 4 Mbytes of shared memory and the VMEbus slave range is to start at address 0xC00000, the following values should be loaded into the CY7C964 registers:

Compare Register: 0x00C00000
Mask Register: 0x003FFFFF

```
/* eval_bd.h includes the following:
   typedef unsigned int WORD
   #define VME_BASE_ADRS 0xE00000 */

#include <eval_bd.h>

#define VMEADRSMASK 0x001FFFFF

WORD *CalcVMEadrs (adrs)
WORD *adrs
{
WORD VMEadrs;

VMEadrs = (WORD) adrs;
VMEadrs &= VMEADRSMASK;      /* mask off upper 11 bits of address */
VMEadrs |= VME_BASE_ADRS;    /* overlay VMEbus address for evaluation board */

return ((WORD *) VMEadrs);
}
```

**Figure 3. VMEbus Address Calculation**

```
/* eval_bd.h includes the following:
   typedef unsigned int WORD */

#include <eval_bd.h>

#define LCRADRSMASK 0xFFE00000
#define LCRMASK 0xE003FFFF
#define LCRSHIFT 3

WORD *CalcLCR (adrs, LCReg)
WORD *adrs, LCReg;
{
WORD TempAdrs;

TempAdrs = (WORD) adrs;    /* convert WORD pointer to WORD */
TempAdrs &= LCRADRSMASK;   /* mask off lower 21 address bits */
TempAdrs >>= LCRSHIFT;     /* shift over by 3 */
LCReg &= LCRAMSK;          /* clear out existing address in LCReg */
LCReg |= TempAdrs;         /* overlay new address onto LCReg */

return (LCReg);
}
```

**Figure 4. LCR VMEbus Address Calculation**

### Compiling Considerations

Because the monitor used for the evaluation board is EPROM-based, certain considerations are noted, namely:

1. All monitor sections that can be read-only are linked such that they occupy a contiguous section of EPROM. This may be done with the -R option of a UNIX *cc* compiler. The -R option merges the code segment TEXT with the initialized data segment DATA.
2. Because the DATA segment is now located in EPROM, any initialized data is now read-only and is not modifiable. This suggests that variable declarations do not initialize the variable, as shown in *Figure 5*.
3. The uninitialized data segment BSS and the stack segment STACK must be located in RAM.

```
/* NO!!! */
WORD *VMEadrs = (WORD *) 0x400000;

/* Yes!!! */
WORD *VMEadrs;
VMEadrs = (WORD *) 0x400000;
```

**Figure 5. Proper Variable Initialization**

## Example VIC64 Software Building Blocks

The following are examples of code that were used for the VIC64-specific routines on the board.

### vic.h

vic.h is a header file that defines useful macros and VIC64-register-related constants. First, the macro VIC is defined, which returns an address to a VIC64 register. The argument to this macro is the number of the register. These numbers start from 0 (VIICR) and end with 57 (BTLR2) for the VIC64 (56 for the VIC068). These numbers are not the address of the register. Next, constants are defined that assign these numbers to the register names themselves. And lastly, a unique VIC64 register identifier is given to each register so that its address and contents can be obtained directly. A similar macro is defined for setting and clearing the Interprocessor Communication (IPC) switches. This IPC macro needs, as an argument, the starting address of the VMEbus IPC areas of interest.

As examples, consider the code fragment shown in *Figure 6*, which illustrates the VIC_xxx macros.

In addition, numerous other constants are defined that aid in manipulating the various bit fields within the registers themselves. These constants are separated by register. Also, the last character of the constant name may consist of a underscore ( _ ) or lower case letters that indicate something about the constant or the bits. *Table 5* summarizes these characters.

```
#include <vic.h>              /* VIC macros located here */
#include <eval_bd.h>          /* typedef for BYTE (unsigned char) */

BYTE      TempStorage;
BYTE      *TempStoragePtr;

TempStorage = *VIC_BTCR;      /* read contents of BTCR */
*VIC_SS0CR0 = TempStorage;    /* store contents of SS0CR0 */
TempStoragePtr = VIC_TTR;     /* read pointer to TTR */
ICF_ICGS0_SET (ICF_BASE);     /* set ICGS0 */
```

**Figure 6. Using the "VIC" Macros**

**Table 5. *vic.h* Constant Preceders**

| Suffix | Meaning |
|---|---|
| _ | Implies a bit field which is cleared |
| r | Implies read-only bit(s) |
| m | Implies a masking value for bit(s) |

**eval_bd.h**

eval_bd.h is a header file that contains board-specific constants. These constants also include the local address map of the board, including those resources described in *Table 6*.

In addition, other types and constants are defined, including individual DUART registers, power-of-2 constants, byte-extraction macros, and some NVRAM macros.

## A Generic Block Transfer Utility

blt_cmd is a generic, command-line driven program that enables the user to perform almost every conceivable block transfer operation using the VIC64 or the VIC068. One notable exception is allowing the VIC64 to interrupt when the block transfer is complete. blt_cmd is meant mainly to be used as a vehicle for board and code testing.

Configuration is provided by the command-line arguments outlined in *Table 7*.

**Table 6. Local Address Symbols**

| Memory Area | Privilege | Symbol |
|---|---|---|
| EPROM | Read/Write | ROM_BASE_ADDRESS |
| Status Register (LCR) | Read-Only | STATUS1_BASE_ADRS |
| Control Register (LCR) | Write-Only | LOCAL_CONTROL_BASE_ADRS |
| DUART | Read/Write | M68681_BASE_ADRS |
| NVRAM | Read/Write | NVRAM_BASE_ADRS |
| 7C964 Mask Register | Write-Only | BILC_M_BASE_ADRS |
| 7C964 Compare Register | Write-Only | BILC_C_BASE_ADRS |
| Interrupt Acknowledge | Read-Only | INT_ACK_BASE_ADRS |
| VIC64 | Read/Write | VIC_BASE_ADRS |
| VMEbus | Read/Write | VME_BASE_ADRS |
| Private SRAM | Read/Write | BANK1_BASE_ADRS |
| Shared SRAM | Read/Write | BANK2_BASE_ADRS |

**Table 7. Command-Line Arguments**

| Argument | Default[1] | Function |
|---|---|---|
| -6 | | Performs D64 transfers (requires VIC64 device). |
| -3 | √ | Performs D32 transfers. |
| -a[address] | 0xC00000 | Sets local starting address for which data will be read, for VMEbus write block transfers or written for VMEbus read block transfers to *address*. |
| -A[value] | Disabled | Sets user-defined AM code that is to used for block transfers to *value*. |
| -b[value] | 0x200 | Sets minimum value for byte count to *value*. If the -ib value is 0 (increment byte count) the fixed byte will be set to *value*. |

**Table 7. Command-Line Arguments** (continued)

| Argument | Default[1] | Function |
|---|---|---|
| -B[value] | 0xFFFC | Sets maximum value for byte count to *value*. Not used if -ib value is set to 0. |
| -cl | | Enables local boundary crossing. |
| -cL | √ | Disables local boundary crossing. |
| -ct | | Enables 2-kbyte VMEbus boundary crossing (implies -cv). |
| -cT | √ | Disables 2-kbyte VMEbus boundary crossing. |
| -cv | √ | Enables VMEbus boundary crossing. |
| -cV | | Disables VMEbus boundary crossing. |
| -d | | Enables the dual-path option but does not perform interleave master cycles (see -p). |
| -D | √ | Disables the dual-path option. |
| -e | | Sets the release mode to RWD. |
| -E | √ | Sets the release mode to ROR. |
| -f | | Enables DRAM refresh. |
| -F | √ | Disables DRAM refresh. |
| -ib[value] | 0 | Set the byte count increment value to: *value* * size of the operand. |
| -ii[value] | 0 | Sets the interleave increment value to *value.* |
| -iu[value] | 0 | Sets the burst count increment value to *value.* |
| -i[value] | 0 | Sets minimum value for interleave to *value*. If the -ii value is 0 (increment increment count) the fixed interleave value will be set to *value*. |
| -I | 0xF | Sets maximum value for interleave to *value*. Not used if -ii value is set to 0. |
| -k | √ | Enables data set-up before every block transfer and data checking after every block transfer. |
| -K | | Disables data set-up before every block transfer and data checking after every block transfer. |
| -l[value] | 1 | Sets the number of block transfers to perform to *value.* If *value* is set to 0, program will loop forever. |
| -m | | Enables the clearing of the BLT enable bit (BTCR[4]) during the first interleave (VIC64 only). |
| -M | √ | Enables the clearing of the BLT enable bit (BTCR[4]) after the block transfer is completely finished. |
| -p | | Enables the dual-path feature and performs VMEbus master cycles during the interleave period. |
| -P | √ | Disables the performing of interleave master VMEbus cycles. Leaves the dual-path feature enabled. |
| -r | √ | Enables BLT reads. |
| -R | | Disables BLT reads. |
| -s[value] | 3 | Sets the VMEbus request level to *value*. |
| -t | √ | Enables the “enhanced” BLT turbo mode (VIC64 only). |
| -T | | Disables the “enhanced” BLT turbo mode. |
| -u[value] | 0 | Sets minimum value for the burst count to *value*. If the -iu value is 0 (increment burst count) the fixed burst count will be set to *value*. |
| -U[value] | 0x3F | Sets maximum value for burst count to *value*. Not used if -iu value is set to 0. |
| -v[value] | 0xDEADC0DE | Sets the value to which destination memory will be set to *value*. |
| -w | √ | Enables BLT writes. |
| -W | | Disables BLT writes. |
| -x | | Restores all options to their default states. |
| [address(es)] | 0x200000 | VMEbus starting address(es) for block transfer. Up to five may be specified. |

**Note:**

1. The check mark indicates the default of two preceding arguments.

All mutually exclusive options are shown without a divider between the options. If two mutually exclusive options are defined, the last one in the command line will take precedence. The state of these options are saved in static variables such that once a configuration is entered, the whole command string will not have to be retyped. Only those options that need to be changed will have a new option. Using the -x option will restore all options to their default state.

**Unsupplied Functions**

blt_cmd.c contains one function, lib_atohex(), that is not supplied. It is a library routine supplied with the SPARCmon source. Any ASCII-to-hex converter could be used with small modifications to blt_cmd.c. lib_atohex() is outlined in *Figure 7*.

**Program Flow**

*Figures 8*, *9*, and *10* illustrate the flow of blt_cmd.c.

**Example Operations**

The following examples show how blt_cmd can be used to initiate a variety of block transfers.

```
blt_cmd -l0 -6 -ii1 -aC800000 D800000
```

This command line would perform D64 read and write block transfers indefinitely using local address 0xC800000 and VMEbus address 0xD800000. After each read/write block transfer, the interleave period is incremented by 1. All other options would remain at their default values.

```
blt_cmd -3 -W -iu1 D800000 E800000
```

This command line would perform D32 read block transfers indefinitely (-l0 still in effect) using local address 0xC800000 (defined last time) and VMEbus addresses 0xD800000 and 0xE800000. After each read block transfer, the burst count and the interleave period (still defined from last time) is incremented by 1. All other options would remain at their default values.

```
blt_cmd -6 -w -ib1 -K -p D800000
```

This command line would perform D64 read and write (writes are re-enabled with -w) block transfers indefinitely using local address 0xC800000 and VMEbus address 0xD800000. After each read/write block transfer, the byte count would be incremented by 8 (1 * 8 bytes/transfer). Data checking is suppressed. Master cycles are performed in the interleave period. All other options would remain at their default values.

```
blt_cmd
```

Performs block transfers using the same parameters as the last time invoked.

```
#include <atohex.h>
/* needed for lib_atohex return
   values */

lib_atohex (string, hexvalue)
char *string;
unsigned long *hexvalue;

/*
inputs:
string    character to be converted
Outputs:
hexvalue  pointer to the hex result

Return value:
SUCCEEDED      valid number
(otherwise)    illegal hex number
*/
```

**Figure 7. atohex() prototype**

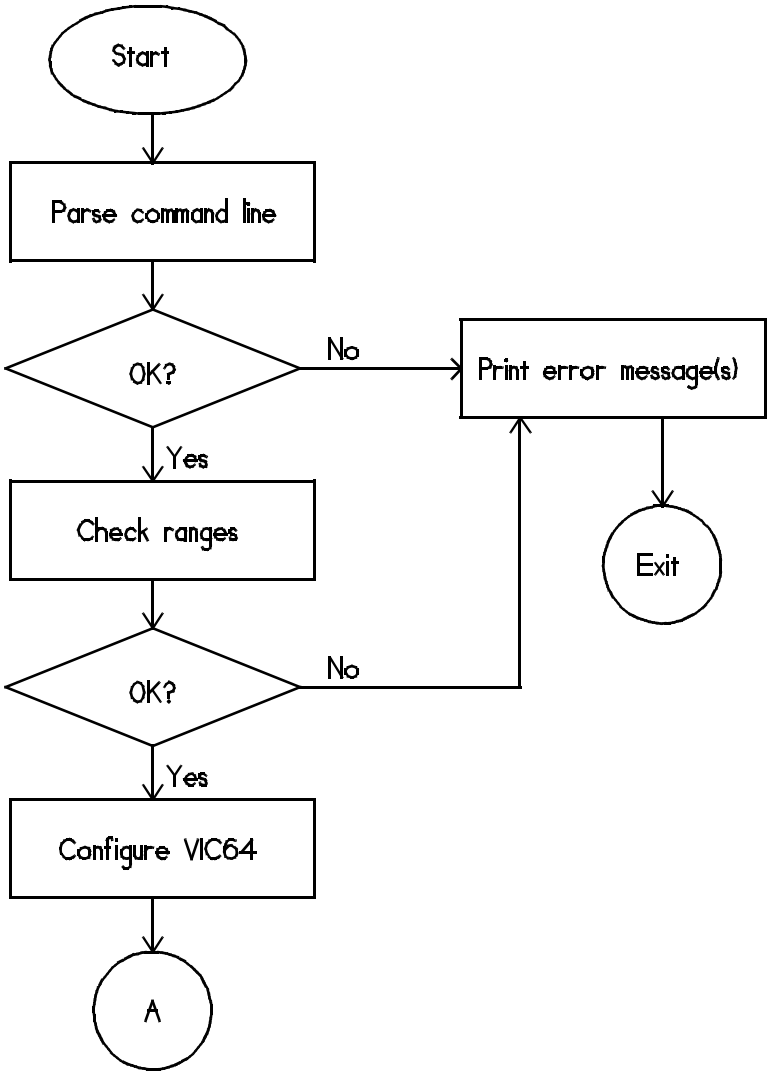


**Figure 8. blt_cmd Flow**

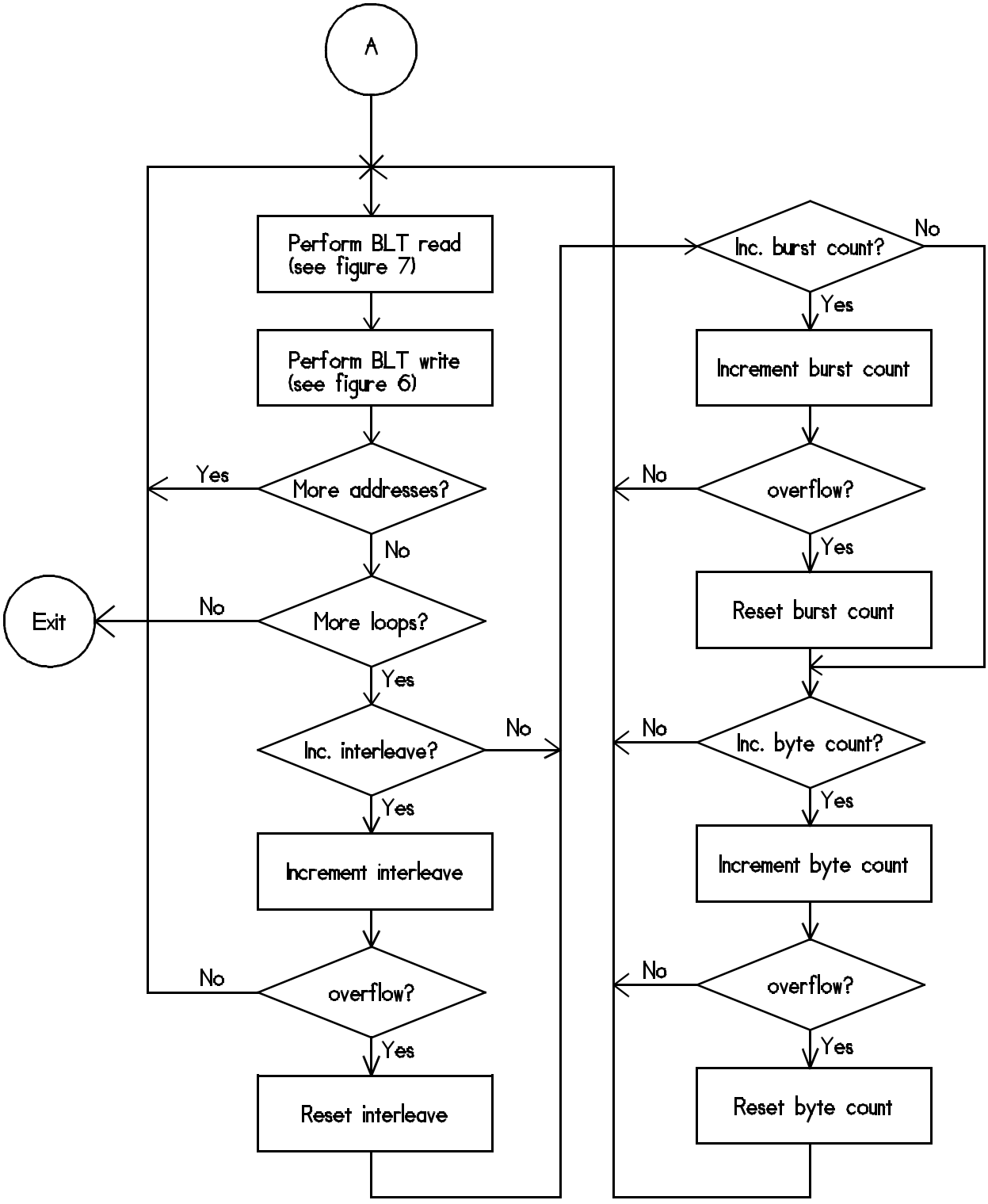


**Figure 8. blt_cmd Flow** (continued)

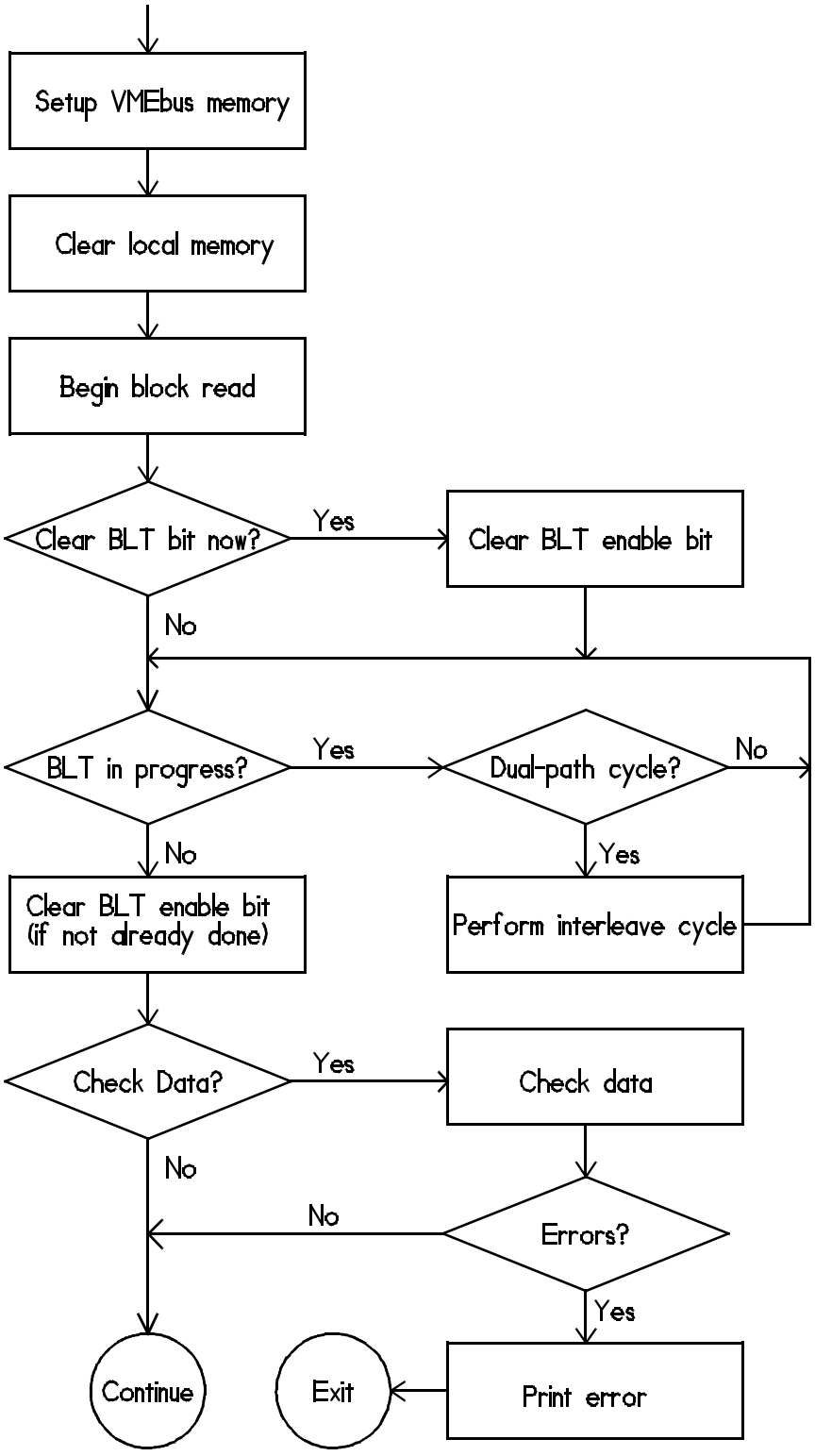


**Figure 9. blt_cmd Read Flow**

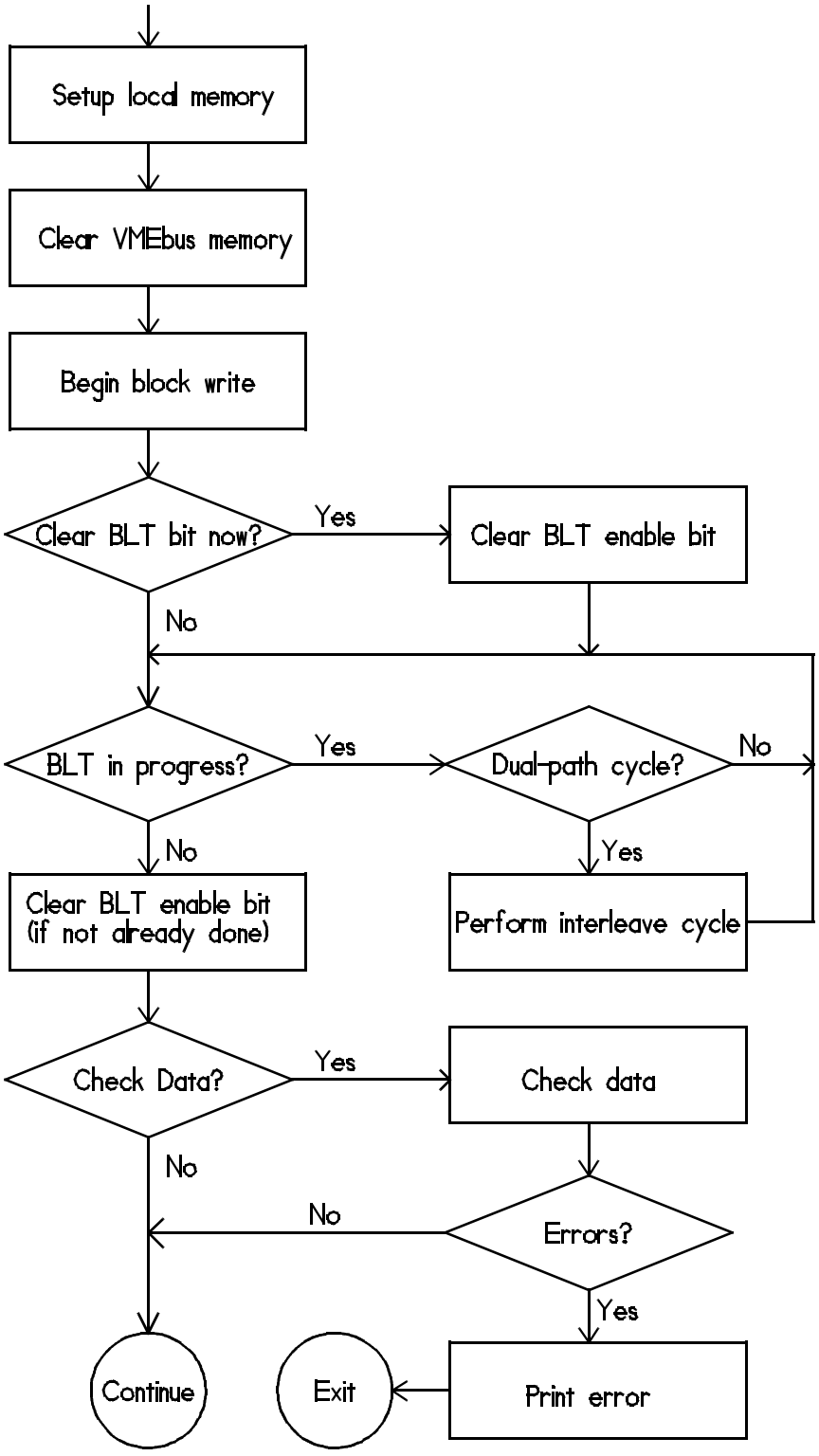


**Figure 10. blt_cmd Write Flow**