File: CBUG.DOK

CBUG – User Manual

Release 3.0
Matthias Wille
29 September 1986

The Monitor CBUG presented in this paper serves as a low-level debugging aid for the workstation Ceres (Computing Engine for Research Engineering and Sience). Ceres is based on the NS32032 processor and has been developped at the Institut für Informatik of ETH. The Ceres hardware is described in [HE]

CBUG is a tool that enables the user to change and display all CPU registers and gives read/write access to the whole address space. The whole chipset comprising the CPU, FPU and MMU is supported by CBUG. The monitor provides a downloading facility to load programs directly into Ceres' memory. Three datatypes are provided, namely byte, word and doubleword. The commands to display and change information in the registers and the memory may be parametrized by these types. A set of fundamental routines dealing with serial and parallel I/O is provided for the user. The following sections describe the commands and the interface of the monitor in detail.

1. Machine Configuration

CBUG works together with the program NServer (see appendix A) running on a Lilith. The NServer serves both as a terminal and a program loader for the Ceres. The interface between the Ceres and the Lilith consists of a RS232 serial interface and an optional 16 bit bidirectional parallel interface. The serial interface is used both as terminal and to load programs while the parallel interface is used only for downloading.

2. General Command Syntax

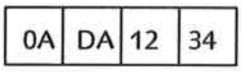
A command is generally of the following syntax.

```
Command          = CommandCode [TypeSpecifier] [" " Parameter {"," Parameter}] [<CR>|]
                 | CommandCode [ImmediatePar].
CommandCode      = "single letter".
TypeSpecifier    = "B" | "W" | "D".
Parameter        = "...".
ImmediatePar     = "single letter".
```

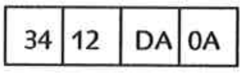
The type specifier immediately follows the command code. It denotes the type of the parameters or operands. Note, that not all commands may be parametrized with a type. The parameters are separated from the command code by a space. The parameters itself are separated by comma unless defined otherwise or the second form of command is used. CBUG makes no distinction between lowercase and uppercase letters.

2.1. Types

The philosophy of the types stoms from the fact that there are two different methods to store the data on the Ceres. The first one is used for the registers while the second one applies onto data in the memory .

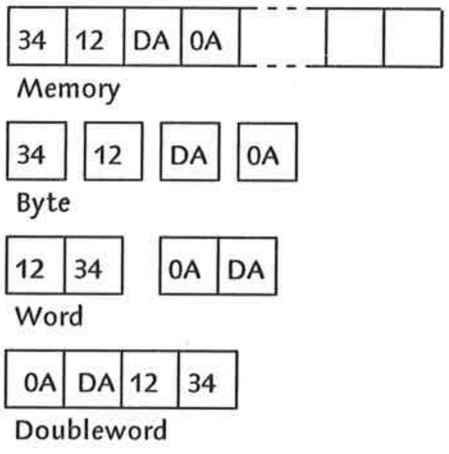


Register



Memory

Both figures show the same 32-bit quantity with the value 0ADA1234H. The first denotation is the one that a programmer thinks of when he writes his program. The most significant byte apears at leftmost position and the least significant byte at the rightmost position. The memory representation is exactly vice-versa. In order to avoid confusion when viewing at the same value in a register and in memory, we propose the types byte, word and doubleword. If one wants to look for a word the representation is the same in both cases. Thus it is quite easy to look e.g. at the dump of a word oriented list or array. No byte swapping in mind is necessary because it is done by CBUG's display routine.



As seen above converting of memory to different types is just a matter of the order of the bytes. Thus the user is responsible to give a proper start address to produce meaningful output. Types may be applied only on data in the memory. Registers are handled as a whole. Most of them are double words except the MOD and PSR registers which are words.

2.2. Expressions and Registers

An expression is a term that yields an effective address to be used as parameter for a CBUG command. The term may be either a hexadecimal value or a valid address expression of the NS32000. For simplification we allow only the modes register relative, memory relative and absolute[NS]. We propose this feature in order to avoid complex hexadecimal computations in mind or on paper when debugging a program.
A register may be every CPU, FPU and MMU register as described in [NS]. In the CBUG all register names longer than two characters are shortened as shown below.

unterstr. zu dick

NS	CBUG
INTBASE	IN - interrupt base register
SP0,SP1	S0, S1 - interrupt and user stack pointers
MOD	MO - module descriptor
PSR	PS - program status register
FSR	FS - floating point status register (FPU)
EIA	EI - Error/Invalidate Address register (MMU)
MSR	MS - MMU Status register (MMU)
PTB0,PTB1	P0, P1 - page table registers (MMU)

The general purpose registers are named R0..R7 and F0..F7. For indexing in expressions only the registers R0..R7, PC, SB, FP, S0 and S1 are allowed.

2.3. Data Representation

The CBUG monitor works with a virtual register file. It is a copy of the CPU registers at the time the program was stopped. All changes of CPU registers refer to the virtual registers while FPU and MMU registers are modified directly. Note, that when the coprocessors are not set in the configuration register (DIP-switch on the CPU board) their registers are inaccessable. Memory inspections and changes are made in the real memory. When a program is started again the virtual registers are copied to the real ones before.

3. Command Description

3.1. Display data

The display command is used to display the contents of specified memory locations or registers. It has the form

```
    D [B|W|D] fromexpression {":"toexpression} <CR> |
    D register.
```

The parameters are separated from the command and the typespecifier by a space char. The first form of the parameters displays a range of memory starting at 'fromexpression' and ending at 'toexpression'. If the 'toexpression' is omitted 16 bytes are shown. The memory is displayed according to the type specifier. The second form shows the contents of a register.

Examples:

D R0	displays the contents of R0
DW 5(R0)	displays the word pointed to by the contents of R0 incremented by the displacement 5
DB FFH:1FFH	displays the bytes from the memory locations FFH to 1FFH

The output of data may be stopped by pressing CTRL-S and restarted by pressing CTRL-Q. If a display command should be aborted CTRL-X has to be issued.

3.2. Change Data

The change command is used to change the contents of specified memory locations or registers. It has the form

```
    C [B|W|D] expression {, value} <CR> |
    C register {, value <CR>}.
```

In the first form of the change command the contents of the location at the address yielded by 'expression' will be replaced by 'value'. If the second parameter is specified the contents of the location is changed without showing the old contents of the location. If the second parameter is omitted the old contents of the location is shown before a new value will be aquired. The new value has then to be entered after typing a colon. The range of the value depends on the specified type. If e.g. byte was specified only the LSB of the value will be used. When the new value is terminated with a "," the next location can be changed according to the given type specifier. A "\" as terminator offers the previous location to change. The <CR> returns control to the command mode, i.e. the next command can be entered. The second form is used to change the contents of a register either with or without showing the old contents.

Examples:

C R0	display the contents of R0 and change if wanted
C S1,0	sets the current user stackpointer to zero, without notice on the old value
CW FF,DADA	changes the word at address FF to the value DADA

3.3. Display Status

The status command will be invoked by typing

```
    S
```

without <CR>. It displays the contents of all registers in the virtual register file and the type of the last exception and were it took place.

3.4. Loading of Programs

In order to cross-test programs developped for Ceres on the Lilith a load command is provided. It uses either the serial or the parallel interface to download programs into memory from the disk of the Lilith. The transfer format is described in the appendix. The command is started by typing

4

LS or LP

without a <CR>. The second letter is an immediate parameter and selects the serial ('S') or the parallel ('P') interface for transmission. The server program on the Lilith now waits for a file name to be typed. The code has to be either on a REL file as produced by the assembler or on an ABS file generated by the cross linker. The code is loaded starting at the addresses transmitted with the code. During the transfer a check for transmission errors is performed. The transferred data contains the start address of the program. This address is stored for a further call to the go command.

3.5. Running a program

The current version of CBUG supports two different kinds of programs namely assembler programs and absolute linked Modula-2 programs. Both may be loaded using the load command mentioned above. They are started using the go-command. It has the form

G {expression} <CR> or
GM.

For both variants the current virtual register file is copied to the processor registers and the control is tranferred to the user program. In the first case either the transferred start address or the one specified in the expression is used to set the PC before starting. In the second case an absolute linked Modula-2 program is started by calling the initialization procedure of the main module of the program, i.e. procedure 0 of the first module.

3.6. Stopping a program

A program running under CBUG may be stopped by software using flag or breakpoint traps. After the program is stopped the contents of all registers is saved to the virtual register file. When the program should be started again at the address where the abort has occured the resume command issued by typing

R

can be used. It restores the saved registers and resumes the execution.

3.7. Exceptions

All possible exceptions on the NS are handled by CBUG although the handlers may be replaced by a user program. On the occurence of a trap or interrupt CBUG saves all CPU registers and restores its own former state. The user is informed about the type of the exception and the module and address where it happened. If the address is within an absolute Modula-2 program it is displayed as a value relative to the start of the modules code. All other addresses are displayed absolute.

3.8. Bootfiles

Assuming that the Ceres fixed disk contains a Medos-2 file system, CBUG provides two commands to handle bootfiles. The commands allow for two bootfiles named PC.BootFile0 and PC.BootFile1 under Medos-2. The first one has a length of 64KBytes and usually contains the Medos-2 operating system. The second file has a length of 96KBytes and may be used either as a backup or for an experimental operating system. The files start at fixed positions and occupy contiguous sectors. As a further possibility to load programs on Ceres, CBUG allows to use the floppy as a boot medium. The following table summarizes the possible files.

tabs

Medium	Name	Sector	Size
fixed disk (normal)	PC.BootFile0	16	64 KByte
fixed disk (alt)	PC.BootFile1	144	96 KByte
floppy	---	5	max. 358 KByte

The command to write boot files is quite similar to that which loads files into memory. It has the form

WS or WP.

As in the load command the second immediate parameter selects the I/O interface which is used to transmit the bootfile. Either the serial ('S') or the parallel ('P') interface may be choosen. After entering the command, CBUG asks for the destination of the bootfile by issueing the message

write boot file
<SP> = normal, <CTL-A> = alt, <CTL-B> = floppy.

The normal bootfile is PC.BootFile0, while alt means the alternate bootfile PC.BootFile1. After having selected the bootfile the transmission starts. The transfer is finished by displaying a message containing the number of blocks written onto the boot medium. A block consists of 512 bytes.

Booting a program is done using the boot command which is initiated by typing

B.

CBUG then asks for the source of the bootfileby writing

load file from
<SP> = normal, <CTL-A> = alt, <CTL-B> = floppy.

When the source is selected the bootfile is loaded and started immediately. Note, that a bootfile must contain an absolute linked Modula-2 program.

Literature:

[HE] Eberle, Hans: The Personal Workstation Ceres, 1986, Institut für Informatik, ETH Zürich

[NS] National Semiconductor Corp.: NS32000 Instruction Set Reference Manual, 1983, Publ. No. 420010099-001A

Appendix A

A. The Lilith Server Program

This section gives a rough description of the Lilith server program NServer. It provides a terminal to the Ceres as well as a facility to load programs from the Lilith disk into the memory of Ceres. The program maintains two windows namly 'dialog' and 'Ceres-Terminal'. The latter is the screen of the Ceres while the first one is used for dialogs on the Lilith and to display status information. The NServer is always in one of the states 'undef', 'terminal', 'load' or 'bootload'. The current state is designated by one of the letters 't', 'l' or 'b' displayed in inverted mode in the dialog window. A certain state is entered from CBUG by sending an appropriate control code via the V24. The code is acknowledged by NServer. The following paragraphs describe the states in detail.

A.1. Terminal Mode

In this mode the NServer behaves as a normal terminal, i.e. all keys pressed at the keyboard are sent to the Ceres and all bytes received from the Ceres are displayed in the Ceres-Terminal window. The receiving includes a flow control protocol using XOFF and XON to stop/start the Ceres sender. This feature is needed because of the limited speed of the Lilith especially when scrolling the screen. The terminal mode accepts all control codes and makes the appropriate switches in the NServer.

A.2. Load Mode and Bootload Mode

In this mode the NServer asks the user for a file to be loaded into the memory of Ceres. The file may be either of type ABS (default) or REL. The first filetype will be produced by the absolute linker the second one by the Ceres cross assembler. While ABS files describe their address itself for a REL file a load address is aquired by NServer. After loading the program CBUG enters the terminal mode. Note that the NServer is unable to act as terminal when in load mode.

Appendix B

B. Transmission Format for Downloading

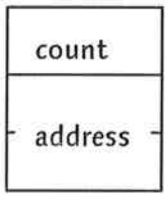
The data can be downloaded using either the 16-bit parallel interface or the serial interface. The actual used by the CBUG must be selected by calling the procedure 'SelDataIO' (see Appendix C). All data except transmission control information is sent within a packet preceeded by a word count and followed by a 16-bit checksum. The checksum is the sum of all words transmitted including the count (mod 216). Each received packet is acknowledged by sending either an ASCII code ACK (acknowledge) or NAK (non acknowledge) within one word back to the sender. After receiving a NAK the sender retries the transmission two times. If all retries fail sender and receiver abort the transmission. The sender in the NServer program sends all words in NS order, i.e. least significant byte first.

B.1. Downloading of Programs

The downloading format is word oriented, i.e. all data sent is aligned to word boundaries. The general transmission format described above is used. The sent file is divided into several blocks. Each block of data is is preceeded by a header block containing the size and address. This is done to prevent from overwriting the memory in case of a wrong received load address. The following paragraphs show the format of the two types of blocks.

B.2. Header Block Format

The header block contains information about the following data block. It has a fixed length of 3 words. The layout is shown in fig. B1.



A header with count zero is interpreted as end of transmission. The address will be used by CBUG as start address for the Go command.

B.3. Data block format

The data block contains as much words as specified in the previous header block. There is no limit on the length.

Appendix C

C. Using the CBUG Software Interface

This section describes with an example how to use the ROM resident CBUG procedures within a user program. The procedures are accesible via a linktable located at a certain position in the ROM. The routines are callable using the CXPD instruction as shown below. The following table gives a summary of the procedures and their entry/exit conditions.

Name	Purpose	Parameters	Registers
InitTerm	initialize terminal	none	R7
Read	read a char	R6> char	none
BusyRead	read a without waiting	R6> char F bit> = 1 got one	none
Write	write a char	>R6 char	none
WriteS	write a string terminated with	0C>R0 string address	none
WriteLn	write newline	none	none
ReadHex	read a hex value	R6> terminator R7> value F bit> = 1 no hex C bit> = 1 overflow	none
WriteHex	write a hex value	>R6 value >R7 type = 0 byte, = 1 word, = 2 double word	none
PIA.Init	parallel interface initialization	none	R7
PIA.Read	read a word from par. interface	R6> word	R6, R7
PIA.BRead	read a word without waiting	R6> word F bit> = 1 got one	R6, R7
PIA.Write	write a word to par. interface	>R6 word	R6, R7
ReadBlk	read a block in general format	>R1 number of words >R2 address F bit> = 1 transmission err.	none
WriteBlk	write a block in general format	>R1 number of words >R2 address F bit> = 1 transmission err.	none
SelDataIO	select transmission interface	>R0 device code = 0 V24, = 1 parallel	none
GetCFG	get configuration register	R0> configuration reg.	none

C.1. Summary of the routine addresses

The following table shows the addresses of the link entries in the CBUG ROM.

```
|**************************************************
| Procedure entry addresses
|
ROMProcs  EQU     $FE0014
InitTerm  EQU     ROMProcs+0*4
Read      EQU     ROMProcs+1*4
BusyRead  EQU     ROMProcs+2*4
Write     EQU     ROMProcs+3*4
WriteS    EQU     ROMProcs+4*4
WriteLn   EQU     ROMProcs+5*4
ReadHex   EQU     ROMProcs+6*4
WriteHex  EQU     ROMProcs+7*4
PIA.Init  EQU     ROMProcs+8*4
PIA.Read  EQU     ROMProcs+9*4
PIA.BRead EQU     ROMProcs+10*4
PIA.Write EQU     ROMProcs+11*4
ReadBlk   EQU     ROMProcs+12*4
WriteBlk  EQU     ROMProcs+13*4
SelDataIO EQU     ROMProcs+14*4
GetCFG    EQU     ROMProcs+15*4
|
|**************************************************
```

C.2. Example

```
|**************************************************
|
          JUMP    Start
|
|**************************************************
| Procedure table
|
ROMProcs  EQU     $FE0014
InitTerm  EQU     ROMProcs+10*4
WriteS    EQU     ROMProcs+14*4
WriteLn   EQU     ROMProcs+15*4
|
|**************************************************
|
Msg.Sampl BYTE    " this is a sample program ",$00
|
Start     LPRD    SP,$1E000       init stack pointer
          CXPD    @InitTerm       initalize terminal
Loop      ADDR    Msg.Hurra,R0
          CXPD    @WriteS         display Msg.Sampl
          CXPD    @WriteLn        display newline
          BR      Loop
|
|**************************************************
```