The Mini-memory module

The mini-memory is a fairly complex, and very usefull, cartridge. It contains a >1800 bytes GROM that maps at GROM address >6000-77FF in all GROM bases, a ROM chip that maps at >6000-6FFF in CPU memory and two static RAM chips that map at >7000-7FFF in CPU memory. The cartridge contains a 3V lithium battery that powers the SRAM even when the cartridge is not plugged in: this way the information contained in the SRAM won't disappear when power is shut down.

The software in GROM and ROM allows you to load and run DV80 assembly language files in tagged-object code format, such as those produced by the Editor/assembler cartridge. A fairly limited, line-by-line assembler is provided on a companion disk, and can be installed in the cartridge SRAM which allows you to create simple assembly language programs. Subprograms in GROM provide support to load and run assembly program from Basic, pretty much like for the Editor/Assembler cartridge.

The GROM also contains a so-called debugger, named Easy-bug, and written entirely in GPL. However, it's more a memory editor (and a poor one at it) than a debugger since it lacks the possibility to set execution breakpoints or to step through an assembly program.

Finally, the cartridge lets you use the SRAM or the memory expansion card as a RamDisk that can contain upto three files.

Hardware

The cartridge consists in 6 chips:

- a 6K GROM (labelled 8231 in my cartridge),
- a 4K ROM (labelled 8234),
- two 2K SRAM 6116 chips mounted in piggy back (i.e. on top of each other, with only the CE* pins connected differently),
- a 72LS322 quad OR gate and

- a 74LS04 hex inverter for the ROM/SRAM selection logic.

In addition, it contains a 3 volts CR2430 lithium battery, with a 3904 NPN and a 3906 PNP transistors (and several diodes) to handle the switching of the SRAM power supply between the console power and the backup battery.

Here's an annotated picture.

And the schematics:

```
  Litium battery     diode
     | +3V          4148
Gnd--||---+-----------|>|--+---------------------------------,
     |    '---|>|---,      |                                 |
             diode  |     |/  3906                           |   6116 SRAM (x2)
   ,---+-----www----+-----|   PNP                            |   +-------+
   |   |      1K          |\  transitor         Gnd---||-----+---|Vcc    |+
   |   |   3904            ^                              WE*>---|WE* A10||---<A5
   |    \| NPN    5.1K     |             DBIN>-----|>o-----------|OE*  A9||---<A6
   |     |---+----www------+----+5V              74LS04    D0>---|D7   A8||---<A7
   |    /|   |             |                               D1>---|D6   A7||---<A8
   |   v     '--www--Gnd   =                               D2>---|D5   A6||---<A9
   |   |        1K         |                               D3>---|D4   A5||---<A10
   |  Gnd                 Gnd                              D4>---|D3   A4||---<A11
   |                                       3.3K            D5>---|D2   A3||---<A12
   '-----------------------------------+---www------,      D6>---|D1   A2||---<A13
                                       '---www---,  |      D7>---|D0   A1||---<A14
                           '04    '32      3.3K  |  |            |     A0||---<A15
                 A4>---+---|>o--,==)>---|>|---+--|--+------------|CE*    ||
                       |        |             |  |            +-------+|
            '04    '32 |        |  +5V---||---'  |         ,------|CE*    |
A3>------+--|>o--,==)>-|--------+       221      |         |      +-------+
         |       |     |        |       4148     |         |
ROMG*>---|-------+     '--------'==)>---|>|------+---------'
         |       | '32           '32              |             Vdd (-5V)
         '-------'==)>------,         +5V---||---'              |
       8234 ROM            |              221 ??               +---www---<RESET*
       +-------+            |                                   |   100 Ohm
 A8>---|1    24|----+5V      |                                   =
 A9>---|2    23|----<A7      |               8231 GROM           |
A10>---|3    22|----<A6      |               +-------+          Gnd
A11>---|4    21|----+5V      |         D7>---|1    16|----Gnd
A12>---|5    20|------------'         D6>---|2    15|----<GR
A13>---|6    19|----<A5              D5>---|3    14|----Vdd(-5V)
A14>---|7    18|----<A4              D4>---|4    13|----<GRC
A15>---|8    17|----<D0              D3>---|5    12|----<DBIN
 D7>---|9    16|----<D1              D2>---|6    11|----<A14
 D6>---|10   15|----<D2              D1>---|7    10|----<GS*
 D5>---|11   14|----<D3              D0>---|8     9|----+5V
 Gnd---|12   13|----<D4                    +-------+
       +-------+
```

GROM wiring

The GROM wiring is fairly straighforward as all the necessary signals are provided by the console through the cartridge port: **D0-D7** make up the data bus, that also serves to set (or retrieve) the GROM address pointer.
GS* is active (low) when a CPU address in the range >9800-9FFF is accessed: it indicates a GROM port access.
A14 serves as an address/data selector: >98x0 is the data port and >98x2 is the address port.
DBIN is used to determine whether a byte should be read or written to the GROM (obviously, only addresses can be written). Note that address line A5 is not needed to differentiate read operations (port >98xx) from write operations (port >9Cxx): this is taken care of by the console when generating the GS* signal.
GRC is the GROM clock, generated by the TMS9918a videoprocessor color burst.
GR is an active high signal indicating that the GROM is ready. When low, it puts the CPU on hold until the GROM is ready.

ROM wiring

Here also, the wiring is straighforward: the cartridge port supplies a ROMG* signals that is active (low) when an address in the range >6000-7FFF is accessed. An OR gate is used to combine this signal with A3 (weight >1000) so that the ROM is active in the range >6000-6FFF.
D0-D7 make up the data bus
A4-A15 make up the address bus (12 bits, i.e. >1000 bytes).

SRAM wiring

D7-D0 are connected to the data bus lines D0-D7. Note the inverse nomenclature: TI considers D0 as the most significant bit, whereas almost everyone else considers it as the least significant byte.
A10-A0 are connected to the address bus A5-A15, providing an address space of 11 bits, i.e. >0800 bytes. Here also, the nomenclature is reversed.
WE* is used to determine whether the memory access is a read or a write operation.
OE* enables the data outputs for read operations. It is controlled by the inverted DBIN line.
The **CE*** connection is the only one which is not wired in the same way on both chips. It is used to determine whether a chip answers to addresses in the range >7000-77FF or to addresses in the range >7800-7FFF. But the selection logic is further complicated by the fact that both pins should be held high when no power is present, so that the SRAMs don't use up the battery trying to output data when the cartridge is not plugged in!

SRAM selection logic

The ROMG* line from the cartridge port signals a memory access in the range >6000-7FFF. It is combined with the inverted address line A3, via an OR gate, to select the range >7000-7FFF. This selection signal is further combined with address line A4 (weight >0800) inverted or not, so as to provide two chip selection signals: one for the range >7000-77FF (piggy-backed chip) and one for the range >7800-7FFF (basal chip). I'm not sure what's the function of the capacitors that connect the CE* pins to Vcc. In fact, I'm not even sure these are indeed caps, they wear five color bands: red-red-brown-silver-silver. Any insight?

Power selection

The two transistors and the associated diodes and resistors are used to select the type of power supply.

When the cartridge is not plugged in, or when it's plugged into a console that's turned off, the battery supplies +3V to the SRAM Vcc pins via a diode. Another diode holds the basis of the 3906 PNP transistor at +3V, which blocks the transistor. The same +3V are applied to the CE* selection pins of the two SRAM chips via a 1K resistor and two 3.3K resistors (one for each chip): this ensures that the chips won't output data when no power is present. Two diodes are used to prevent these +3V from flowing back to the selection logic.

When the cartridge receives power from the console, +5V are applied to the basis of the 3904 NPN transistor via a 5.1K resistor, which makes it passing. As a result, the basis of the 3906 PNP transistor is now grounded through a 1K resistor and the NPN transistor. The +5V current can thus flow through the PNP transistor to the SRAMs Vcc pins (and that's why a diode was placed in the battery circuit: we don't want these +5V to flow back to the battery).

In addition, the 3904 NPN transitor also grounds the two CE* selection pins, via the 3.3K resistors. These resistance values are chosen so that the selection logic (i.e. the OR gates) can easily drive the CE* pins high or low.

Additional circuitery

The RESET connection in the cartridge port is connected via a 100 Ohm resistor to the Vdd power supply and grounded through a capacitor. When the cartridge is plugged in, the RESET line is briefly grounded until the capacitor is charged. This sends a reset signal to the TIM9904 clock generator, which in turns resets the TMS9900 microprocessor, the VDP and the TMS9901. In other words, inserting the cartridge resets the TI-99/4A.

Finally, by-pass capacitors are installed to filter statics by connecting the Vcc pin of each chip to the ground. These caps are labelled 104, which I suppose means that they are 10,000 pF. On the above schematic, only the one for the SRAM chips has been drawn.

Software

The GROM has a standard header which contains:

- Two programs (MINI MEMORY and EASYBUG),
- Three DSRs (MINIMEM, EXPMEM1 and EXPMEM2),
- Seven subprograms (INIT, LOAD, LINK, PEEK, PEEKV, POKEV and CHARPAT).

You'll find a commented disassembly listing of the GROM in these three files: mmg.txt (main menu, DSRs and subprograms), mmg2.txt (CHARPAT, options 1-3 of the main menu, subroutines for internal use), and mmg3.txt (Easybug).

For a commented disassembly of the ROM, look in the files mmr.txt (loader and other assembly subprograms), and mmr2.txt (subprograms dealing with Basic).

Mini memory program

Initialization

When entered, the program loads the built-in character sets from the console GROM:

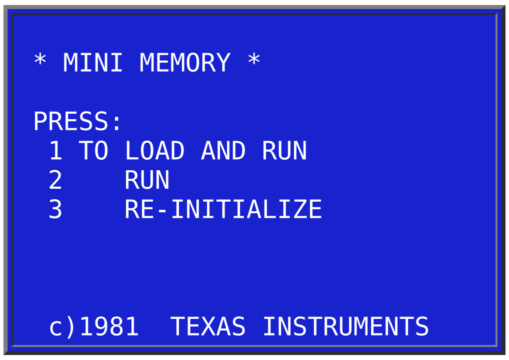
- small upper-case chars 32-95 at VDP address >0900-0AFF,
- lower-case characters 96-127 at VPD address >0B00-0BF7,
- empty chars 0-31 at VDP address >0800-08FF, except for:
- copyright character (10) at address >0850-0857,
- editor cursor (a solid 8x3 pixels vertical bar) at address >08F0-08F7 (i.e. char 30),
- module cursor (an empty 6x6 square) at address >08F8-08FF (i.e. char 31).

For some reason, provision is made to load a different set of characters, possibly corresponding to another console version: the distinction is made by scanning the keyboard with keyboard type 5. If the type is not changed to 0 by the scanning routine, characters >32-95 will be loaded at addresses >0A00-0BFF in VDP memory, then characters 64-95 are modified by adding a small 2-pixel dash in their upper left corner. I have no idea what this is meant for, the TI-99/4 may be?

The VDP registers are then set with the following values:

R1=E0 Standard mode
R2=00 Screen image at >0000
R3=0E Color table at >0380
R4=01 Char pattern table at >0800
R5=06 Sprite attribute table at >0300
R6=00 Sprite pattern table at >0000 (not the same as char pats!)
R7=F5 White on light blue

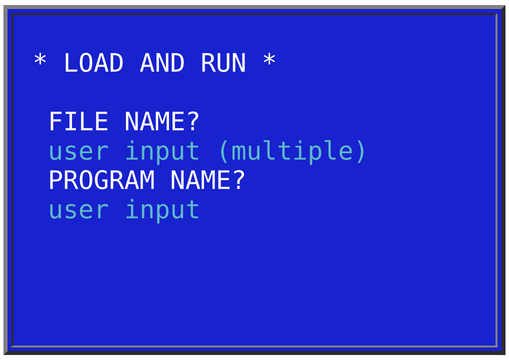
Main menu



```
 * MINI MEMORY *

 PRESS:
  1 TO LOAD AND RUN
  2    RUN
  3    RE-INITIALIZE



  c)1981  TEXAS INSTRUMENTS
```

Option 1 is used to load an assembly file in tagged-object code, it enters option 2 when done.
Option 2 is used to run a program loaded in memory with option 1.
Option 3 is used to reinitialise the SRAM, i.e. to reset the flags and pointers necessary for option 1.
Fctn-9 resets the TI-99/4A.

1. Load and run



```
 * LOAD AND RUN *

  FILE NAME?
  user input (multiple)
  PROGRAM NAME?
  user input
```

Option 1 first checks whether the loading pointers were initialised, by looking for a flag value of >A55A at address >7000. If that's not the case, the pointers are initialised now. The user is then prompted for one or more filenames. These must be Dis/Fix 80 files that contain tagged-object code. Such files are produced by the assembler in the Editor/Assembler cartridge for instance. The program prepares a PAB for the file at address >1000 in VDP memory, with a data buffer area located at >1080. It then issues a XML >71 instruction to call a loader located in the cartridge ROM (vector in >6012). This is the very same loader than the one in the Editor/Assembler cartridge, with one exception: it can handle three different loading areas: the cartridge RAM, the low memory expansion and the high memory expansion. The loading order is: cartridge first, then high-memory, then low-memory.

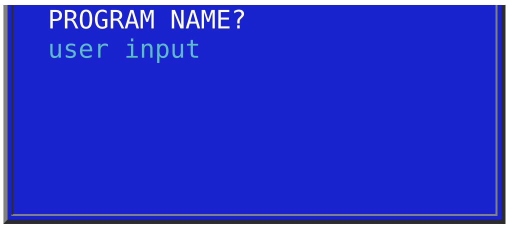
The symbol table for labels DEFined or REFerenced by the loaded file is located at the end of the cartridge RAM, but there also is a table of predefined labels at the end of the ROM, addresses >6F0E-6FFF. It contains the same labels than the editor assembler cartridge, but the standard assembly routine are located in the cartridge ROM instead of the low memory expansion.

The loader can return with an error by setting the Cnd bit in the GPL status byte and placing the error code in byte >8322. The module will then issue an error message and (in most cases) re-enter the load-and-run menu.

If all went OK, the user is prompted for another filename, just as above. The loader is then called again and the whole process repeats until the user enters an empty filename, at which point the program moves automatically to option 2.

2. Run

```
  * RUN *
```



```
PROGRAM NAME?
user input
```

This menu can either be entered from the main menu, which displays the "` * RUN * `" title, or from the "load-and-run" menu, after the user entered an empty filename.

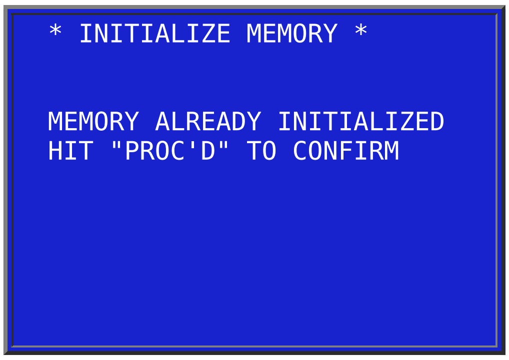
The user is prompted for a 6-character program name. This must correspond to an assembly label, i.e. contain only upper-case letters and digits, the first character cannot be a digit. This label should be present in the symbol table, either because it's part of the pre-defined table or because it was placed here by the loader (as a result of an assembly-language DEF statement). It is legal to enter an empty program name, in which case the last run program will be re-entered (if any).

The module checks for a flag value of >A55A in word >2000, indicating that the standard assembly language routines were dumped from the GROM. If the value is not found, a "`PROGRAM NOT FOUND`" error is issued.

Then the module sets up the VDP for the program to be executed: the color table is filled with >13 bytes (black on light green), the screen is erased, and VDP register 7 is set as >F3 (white on light green). Finally, the linker is entered via an XML >70 (i.e. vector in >6010). Upon return, the Cnd bit in the GPL status byte will cause an error message, if it's set. The error code is taken from byte >8322, that was cleared before calling the linker. Otherwise, the VDP setup of the module is restored, and the message ""`PRESS ENTER TO CONTINUE`" is displayed at the bottom of the screen.

When the user presses <enter> the program returns to the main menu. Note however that nothing is erased in memory, so the user can select option 4 again and run another program, or the same one. To run the same program again, the user can just enter an empty program name.

3. Initialize memory



```
 * INITIALIZE MEMORY *


MEMORY ALREADY INITIALIZED
HIT "PROC'D" TO CONFIRM
```

This options first checks for the presence of a >A55A flag at address >7000. If it was found, the SRAM was already initialized and there may even be a program loaded in it. The program therefore asks the user for confirmation: pressing Fctn-6 resets the loading pointers, installs the flag, and wipes out the rest of the SRAM. Any other key returns to the main menu without any changes.

If the flag is not detected (or if Fctn-6 is pressed), the program first clears the whole SRAM range (>7000-7FFF), then tests for the presence of a memory expansion card by writting and reading back address >A000. It then sets the loading pointers accordingly at addresses >7022-702D (or only >701C-701F if no memory expansion is present).

Cartridge RAM usage

```
Address Initially Usage 
>7000   >A55A     Flag: loader pointers were initialised
```

```
>7002-7011        Used by CALL LINK to store parameter types

>701C   >7118     First free address in cartridge RAM: FSTMOD
>701E   >8000     Last free address in cartridge RAM: LSTMOD(bottom of symbol table)
>7020   >0000     Previous program address
>7022   >A000     First free address in high memory: FSTHI
>7024   >FFE0     Last free address in high memory: LSTHI
>7026   >2000     First free address in low memory: FSTLOW
>7028   >3FFF     Last free address in low memory: LSTLOW
>702A             Checksum calculated by the loader for each record
>702C             Used by LOADER to save the PAB pointer
>702E             Used by linker to save its return address
>7030             CRU address where DSR was found (set by DSRLNK, used by loader)
>7032             DSR address in card ROM (used by LOADER to speed up file operations)
>7034             DSR name size (copy of >8354)
>7036             DSR name ptr (copy of >8356)
>7038             Number of DSR occurences found (copy of GPL R1)

>708A-7091        Buffer for DSR name (for quick comparison by DSRLNK)
>7092-70B1        Workspace for most standard assembly routines
>70B8-70C7        User-workspace upon program entry (return address in R11)
>70D8-70F7        Workspace for the linking-loader
>70F8-7117        Workspace for Basic-handling subroutines
```

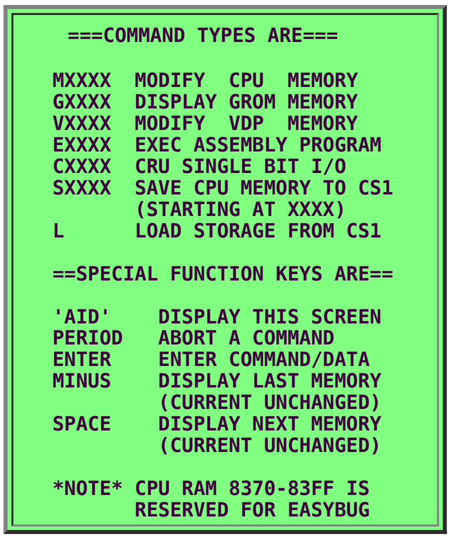
Pre-defined symbol table (>6F0E-6FFF)

Label	Value	Usage
'UTLTAB'	>7020	Utility table for loader
'PAD '	>8300	Scratch-pad memory
'GPLWS '	>83E0	GPL workspace
'SOUND '	>8400	Soud port
'VDPRD '	>8800	VDP read-data port
'VDPSTA'	>8802	VDP status-read port
'VDPWD '	>8C00	VDP write-data port
'VDPWA '	>8C02	VDP write-address port
'SPCHRD'	>9000	Speech synthesizer read-data port
'SPCHWT'	>9400	Speech synthesizer write-data port
'GRMRD '	>9800	GRAM/GROM read-data port
'GRMRA '	>9802	GRAM/GROM read-address port
'GRMWD '	>9C00	GRAM write-data port
'GRMWA '	>9C02	GRAM/GROM write-address port
'SCAN '	>000E	Keyboard scanning routine in console ROM
'XMLLNK'	>601C	Subroutines to call an XML from assembly (in cartridge ROM, WS >7092)
'KSCAN '	>6020	Subroutine to call the SCAN subroutine (in cartridge ROM, WS >7092)
'VSBW '	>6024	VDP single byte write subroutine (workspace >7092)
'VMBW '	>6028	VDP multiple bytes write subroutine (workspace >7092)
'VSBR '	>602C	VDP single byte read subroutine (workspace >7092)
'VMBR '	>6030	VDP multiple bytes read subroutine (workspace >7092)
'VWTR '	>6034	VDP write to register subroutine (workspace >7092)
'DSRLNK'	>6038	Subroutine to call a DSR or a subprogram (workspace >7098)
'LOADER'	>603C	Tagged-object code files loader (workspace >70D8)
'GPLLNK'	>6018	Subroutine to call a GPL subroutine from assembly (workspace >7092)
'NUMASG'	>6040	Subroutine to assing a value to a Basic numeric variable (WS >70F8)
'NUMREF'	>6044	Subroutine to get the value of a Basic numeric variable/constant
'STRASG'	>6048	Subroutine to assing a value to a Basic string variable
'STRREF'	>604C	Subroutine to get the value of a Basic string variable/constant
'ERR '	>6050	Subroutine to re-enter Basic with an error message

EasyBug program

EasyBug is a fairly primitive debugger, written entirely in GPL (apart for a few lines of assembly) that can therefore be run from the cartridge GROM without affecting the program it is supposed to debug. Its main drawback is that it does not easily allow to set breakpoints, nor to execute a program step by step. In fact, it is a line-by-line memory editor than a debugger...

Upon entry, EasyBug loads the patterns for upper case characters 32 to 96 at VDP address >0900. It then display the following help screen and waits for the user to press a key.


```
    ===COMMAND TYPES ARE===

  MXXXX  MODIFY  CPU  MEMORY
  GXXXX  DISPLAY GROM MEMORY
  VXXXX  MODIFY  VDP  MEMORY
  EXXXX  EXEC ASSEMBLY PROGRAM
  CXXXX  CRU SINGLE BIT I/O
  SXXXX  SAVE CPU MEMORY TO CS1
         (STARTING AT XXXX)
  L      LOAD STORAGE FROM CS1

  ==SPECIAL FUNCTION KEYS ARE==

  'AID'    DISPLAY THIS SCREEN
  PERIOD   ABORT A COMMAND
  ENTER    ENTER COMMAND/DATA
  MINUS    DISPLAY LAST MEMORY
           (CURRENT UNCHANGED)
  SPACE    DISPLAY NEXT MEMORY
           (CURRENT UNCHANGED)

  *NOTE* CPU RAM 8370-83FF IS
         RESERVED FOR EASYBUG
```

Once a key is pressed, easybug clears the screen, displays a question mark on the last line and waits for a command. No cursor is displayed.

To enter a command, the user must type one of the following keys:
M to edit CPU memory,
V to edit VDP memory,
G to read GROM memory,
C to read and write CRU bits,
S to save CPU memory on tape,
L to load the content of a tape file in CPU memory,
E to execute an assembly program (in CPU memory),
Fctn-7 (Aid) to display the help screen,
Fctn= (Quit) to leave EasyBug and reset the TI-99/4A.

Most commands must be followed by an hexadecimal address. Valid keys to enter an address are 0 through 9, A through F, <enter> that starts the command and the decimal point that aborts it and returns to command mode. There is no backspake key available, but if you make a mistake you can just keep typing: only the last 4 digits will be considered.

The M command

This commands is used to display and edit CPU memory (obviously, ROM memory cannot be edited). Easybug displays the address, an equal sign, and the content of this address as an hexadecimal byte. An arrow follows, after which you can enter a new value for that byte.

To move to the next address, press the spacebar. Any value you may have entered is ignored.
To move to the previous one, press the minus key. Any value you may have entered is ignored.
To abort the command and return to the command line, press the decimal point.
To abort the command and go back to the help screen, press the < key (why not Fctn-7, I have no idea).

To modify the content of a memory address, enter an hexadecimal byte and press <enter>. Valid digits are 0 through 9 and A through F. Just as above, no backspace key is provided but errors can be "corrected" by just typing the correct value again: only the last 2 digits are considered. Pressing <enter> with no new value leaves the memory byte unchanged (and is thus equivalent to pressing the spacebar).

```
 ?M2000
  M2000 =FD ->
  M2001 =00 ->
  M2002 =FF ->
  M2003 =00 ->55
  M2004 =00 ->
 ?M2003
  M2003 =55 ->
```

Note that EasyBug won't let you modify the scratch-pad addresses >8370-83FF as they are used by the program (and by the GPL interpreter). And of course, you can't modify the content of the console ROMs, although EasyBug will let you try.

Interestingly, the read and write operations are not performed by the GPL instruction ST, as one may have expected. Instead, a tiny assembly program is loaded in the scatch-pad, at addresses >83B0->83BF and executed via a XML >F0 (which means the vector is at >8300, but EasyBug saves the content of this address into >838E, so it can be restored upon return from assembly.

```
* Routine that reads a byte from CPU memory. The GPL program places
* the address in >839A and expects the result in >8391.
       MOV  @>838E,@>8300     restore vector location (saved by GPL)
       MOV  @>839A,R1         get address
       MOVB *R1,@>8391        read a byte
       B    *R11              return to GPL

* The same routine is loaded and slightly patched to modify a byte.
* It expects the new byte value in >8399.
       MOV  @>838E,@>8300     restore vector location (saved by GPL)
       MOV  @>839A,R1         get address
       MOVB @>8399,*R1        write byte to address
       B    *R11              return to GPL
```

The V command

This command lets you view and edit the content of the VDP memory. It is used just as the M command. Note that fooling around with the screen area (addresses >0000-0300) or the character pattern table (addresses >0900-0AFF) may render the screen difficult to read...

Normally, the valid range of VDP addresses is >0000-3FFF, but there is an interesting bug in the GPL program. Instead of masking the address you entered with a AND >3FFF, EasyBug ANDs it with >FFFF... which does strictly nothing. This means that you can modify the VDP register by entering an address in the form >8rxx, where r is the register you want to modify and xx the new value. Note that modifications to registers 0, 1, 2, 3 and 4 may result in making the display unreadable...

Examples:

```
 ?V87F5            changes the screen color to blue
  V87F5 =38 ->     press . here

 ?V81F0            places the screen in text mode
  V81F0 =38 ->     press .
```

Note that the initial value of register 1 is automatically restored when a key is pressed, using the content of byte >83D4 (which EasyBug won't let you modify).

The G command

This command lets you view the content of the GROM memory. It works pretty much like M and V, except that EasyBug won't make any attempt to modify a byte (although you can enter a new value, it is just ignored). Thus, you cannot use EasyBug to patch a program in a GRAM card.

In addition, no provision is made to change the GROM base: the default base found in word >83FA-83FB is always used (and EasyBug won't let you modify >83FA).

```
 ?G6000
  G6000 =AA ->
  G6001 =01 ->
  G6002 =00 ->
  G6003 =00 ->55
  G6004 =00 ->
 ?G6003
  G6003 =00 ->
```

The C command

This command lets you read and write CRU bits, a very interesting feature that most memory editors don't provide. It works pretty much like the memory edition commands above, with the exception that valid values can only be 0 and 1 (since we're dealing with bits, not bytes). If you enter a greater value, only the least significant bit is considered, e.r. >03 is writen as "1".

Note that according to the device you are accessing, some CRU bits can be read-only bits, other write-only bits (which generally means they read as 0), and some may have different meaning when read or written (which means you don't read back what you wrote).

The CRU address is the value you would normally place in R12 for CRU operations from assembly. For instance, >1100 to access bit 0 in the disk controller card. Since A15 is used to carry the data, only even addresses are valid. However, EasyBug will read bit from odd addresses (the bit will be the same as in the even address) and write bit from them (bit "1" will be writing first, followed by the value you selected).

Caution: be carefull when modifying CRU bits, as it is easy to lock-up the system. Have a look at my CRU page for more information about the CRU.

EasyBug performs CPU output with the GPL instruction I/O 3, but uses a small assembly program to perform CPU input (beats me). This program is loaded and executed in the scratch-pad, at addresses >83B0-83C1.

```
* This routine reads a CRU bit into byte >8391
* The CRU address is expected in word >839A
      MOV  @>838E,@>8300     restore vector location
      MOV  @>839A,@>83F8     load address in R12
      STCR @>8391,1          set 1 CRU bit
      B    *R11              return to GPL
```

The E command

This command lets you execute an assembly program, starting at the address you entered. No provision is made for EasyBug to regain control via breakpoints or interrupts. It is up to the executed program to return to GPL with either `B *R11` or `B @>006A.` Note that the GROM address should not be altered (or else it should be restored after execution).

This make the Execution command of limited interest for debugging purposes. Mainly, it is used to run a program loaded with the L command.

To enter the selected assembly program, EasyBug uses the same assembly routine than to patch the CPU memory (cf the M command), after patching one instruction:

```
* This routine starts execution of an assembly program.
* The start address is expected in word >839A
      MOV  @>838E,@>8300     restore vector location (saved by GPL)
      MOV  @>839A,R1         get address
      B    *R1               patched: branch to start address
```

Example:

We could use the E command together with the M command to patch bytes in the memory area tha EasyBug won't let us modify. For instance, we could modify byte >83D4: it contains a copy of VDP register 1 that is used by the keyboard scanning routine to restore VDP register 1 each time a key is pressed.

```
?M8302
 M8302 =00 ->D8     This is MOVB @>830A,@>83D4
 M8303 =00 ->20
 M8304 =00 ->83     Here is the address to modify
 M8305 =00 ->0A
 M8306 =00 ->83
 M8307 =00 ->D4
 M8308 =00 ->04     This is B *R11
 M8309 =00 ->5B
 M830A =00 ->F0     This is our new value
 M830B =00 ->       Press . here
?E8302              Execute our routine
?                   Press any key to place the screen in text mode
```

The same routine could be used to change the GROM base in byte >83FB. If you are running EasyBug from your GramCard, you can now access different GROM ports in your card. Just make sure that the mini-memory GROM is copied in each one, otherwise EasyBug will crash.

The S command

This command allows you to save a program from CPU memory onto the CS1 cassette recorder. Unfortunately, it won't work with disk drives.

The address you enter is the first address saved, and you are then prompted for the last address to save. EasyBug verifies that the end address is higher than the start address, and that they do not lie in the console ROM (address lower than >2000), nor encompass the area reserved for EasyBug (>8370-83FF). Then EasyBug loads a small assembly program in the scratch-pad addresses >83AA-83BF and executes it with an XML >F0. This program just copies the selected memory range into a buffer in VDP memory at address >1100. EasyBug itself places the program address and size at the beginning of the buffer.

```
* Routine that copies CPU memory to VDP memory.
* It expects the CPU address in word >839C and the
* number of bytes in word >8390.
* The GPL programs sets the VDP address for writing operations.
      MOV  @>838E,@>8300     restore vector location
      MOV  @>839C,R2         get byte pointer
LP1   MOVB *R2+,@>FFFE(R15)  write byte to VDP
      DEC  @>8390            decrement counter
      JNE  LP1               more to do
      B    *R11              return to GPL
```

Finally, EasyBug installs a PAB for CS1 in VDP memory, at address >1000-100D and calls the DSR scanning routine in the console GROMs at G@>0010. The file on tape will be Dis/Fix 64, as required for cassette operations. Note that the PAB specifies a screen bias of >00 in its 8th byte, since we are not using the Basic bias.

The CS1 DSRs, in the console ROM, display all necessary prompts and handle the file operations. Note that EasyBug saves the scratch-pad area >8340-836F before to call the DSR and restores it afterwards. This is necessary because the CS1 DSR modifies several bytes in this area.

The L command

This command is used to load a program previously saved with the S command. It uses the first 4 bytes in the file to determine the loading address and the number of bytes to save.

Here also, EasyBug calls the CS1 DSR to perform the file operations. It then loads the same assembly routine as above, and patches it so that it now copies the VDP memory into CPU memory:

```
* Routine that copies VDP memory to CPU memory.
* It expects the CPU address in word >839C and the
* number of bytes in word >8390.
* The GPL programs sets the VDP address for reading operations.
      MOV  @>838E,@>8300     restore vector location
      MOV  @>839C,R2         get byte pointer
LP1   MOVB @>FBFE(15),*R2+   read byte from VDP (patched)
      DEC  @>8390            decrement counter
      JNE  LP1               more to do
      B    *R11              return to GPL
```

Subprograms

INIT

This subprogram initialises the cartidge RAM for use with by the loader. It is equivalent to option 3 in the Mini-memory main menu.. It wipes out the whole RAM and resets the loading pointers.

LOAD

This subprogram can actually perform two distincts functions: place numeric values at a given address in cpu memory, or call the loader and load a tagged-object file into memory. Which function is selected depends on the type of parameter passed to the subprogram: a (non-empty) string constant or string variable calls the loader, a number or a numeric variable patches the memory.

When a filename is passed as an argument, e.g. CALL LOAD("DSK1.MYFILE"), the subprogram loads the standard assembly language routines then calls the loader with XML >71. Upon return, if the Cnd bit of the status byte is set, an error will be returned. Otherwise, LOAD resumes parsing the parameters. This means you can load several files, or mix file loading and memory patching (however, once you started memory patching, you cannot load any more files).

To load values into memory, you must first specify the destination address as a number from -32768 to +32767, where negative numbers correspond to addresses in the range >8000-FFFF. LOAD calls XML >12 to convert the real number into an integer. The following parameters must contain the bytes to place in memory, therefore the usefull range is 0-255. Note that you can specify any value from -32768 to 32767, but only the least significant byte of the integer will be loaded.

If you want to specify a new address in the same CALL LOAD statement, you can use an empty string constant (or variable) as a separator. The string must be empty so that it is not mistaken for a filename: non-empty strings issue a "bad argument" error.

Example:

```
CALL LOAD("DSK1.MYFILE","DSK2.HISFILE",-24576,65,66,"",-20480,0,218,)
```

Loads a tagged-object code file called MYFILE into memory,
loads the file HISFILE into memory,
places bytes 65 and 66 (i.e. "AB") at location >A000,
and places bytes 0 and 218 at location >B000.

POKEV

Works exactly like LOAD, except that the target is the VDP memory. The address should therefore be in the range 0-3FFF (i.e. 16383). Of course, as programs can't run in VDP memory, it is not allowed to pass a filename within a CALL POKEV.

PEEK

This subprogram performs the opposite of LOAD: it fetches bytes from memory and transfers them in numeric variables.

Just like LOAD, you must specify an address in the range -32768 to +32768 either as a number or as a numeric variable. Then you can enter as many numeric variables as you like: each will receive one byte corresponding to the content of the cpu memory, starting at the specified address. Values will be in the range 0-255.

You can use an empty string as a separator to enter a new address, but non-empty strings will cause an error.

Example:

```
CALL PEEK (12,CLOCK,"",72,A,B,C,D)
XOP2WR=(256*A)+B
XOP2PC=(256*C)+D
```

Gets the content of byte >000C into the variable CLOCK (this is the console clock frequency: 48 for 3 MHz or 40 for 2 MHz).

Then gets the vectors for extended operation 2 (which does not exist in all consoles). The next two statements combine the bytes into words: XOP2WR should be 33696 (i.e.>83A0) and XOP2PC should be 33536 (>8300).

PEEKV

Does the same as PEEK, but gets values from the VDP memory. Addresses should therefore be in the range 0-16383 (i.e >3FFF).

LINK

This subprogram is used to begin execution of a program loaded with CALL LOAD. The first argument should be a quoted string, or a string variable containing the program name (i.e. an assembly DEFined label).

Optionally, you can specify upto 16 additional parameters that will be passed to the executed programs. LINK fetches each parameter from the Basic symbol table, concatenates strings connected with & if necessary, and places the resulting numeric value or a string pointer on the value stack in VDP memory. These values can be retrieved by the called program via XML >18.

In addition, LINK places the total number of variables in byte >8212 and indicates the type of variable (number, string constant, numeric variable, etc) into bytes >7002-7011, one byte per parameter. Note that this is different from CALL LINK in Extended Basic, that places the parameter types at >8300 and from the Editor/Assembler cartridge that uses the low memory expansion for this purpose.

The parameter type values can be:
0: number
1: quoted string
2: numeric variable
3: string variable
4: numeric array element
5: string array element

All this makes parameter handling much easier for programs called from Basic. It's not really a piece of cake though...

Once all parameters have been processed, LINK calls the linker via XML >70, with the program name in >834A-834F and the name length in word >8350-8351. If the name was en empty string, the linker will attempt to call again the previously executed program, if any.

Upon return, ar error is issued if the Cnd bit is set in the GPL status byte. The error number is taken from byte >8322. If no error occured the subprogram removes all parameters from the value stack, if that wasn't done by the called program, and checks that the Basic statement terminates correctly (i.e >8342 must contain >00). Then it returns to Basic.

CHARPAT

This subprogram is used to retrieve the pattern of a character, in the range 32-159.

The first parameter must be a number or a numeric variable: this is the number of the character of interest. The second parameter must be a string variable. Upon return, it will contain an hexadecimal description of the pattern, just like the ones used by the Basic CALL CHAR.

Example:

```
CALL CHARPAT(97,A$)
```

A$ now contains the pattern for character 97, the lower case A: "00000038447C4444"

DSRs

The GROM contains three identical DSRs: MINIMEM, EXPMEM1 and EXPMEM2. Each of them allows you to save a file into a RAM domain, the cartridge RAM, the low memory expansion and the high-memory expansion respectively. They provide you with some kind of very primitive RAMdisk: only 3 files can be saved, and two of them will be forgotten when the power is turned off!

Each memory starts with a 8-byte header containing file info:

```
Offset Contents
0000   >5AA5 flag
0001   file type (>FF for program files)
0002   record length
0004   write pointer (var: mem offset / fix: rec #)
0006   read pointer (ditto)
```

The available memory space is thus:

```
>7008-7FFF for MINIMEM (i.e. >0FF8 bytes)
>2008-3FFE for EXPMEM1 (i.e. >1FF7 bytes)
>A008-FFE0 for EXPMEM2 (i.e. >5FD8 bytes, the rest is reserved for XOP1 and LOAD interrupts)
```

If the loader is intalled in the cartridge RAM (flag >A55A at >7000), and a file is loaded in the memory expansion card, the corresponding pointers are cleared, which prevent the loader from placing programs in this area.

The supported opcodes are Open, Close, Read, Write, Rewind, Load, Save, Delete, and Status. They respect the conventions established for the disk controller.

Open

Performs the necessary file type checks and issues errors if needed (for instance: opening an empty memory for input, or opening a Fix file in append mode). The default record length is 80

If needed,Open initialises the selected memory area, by installing the 8-byte header. It mercilessly overwrites any file currently in memory, without any warning. There is a bug in this routine that causes EXPMEM1 to malfunction if the Mini-memory was initialized.

Close

Returns immediately, without performing anything.

Read

Reads a record from an opened file

Write

Writes a record to an opened file

Rewind

Points to a given record in an opened Fix file or to the top of a Var file.

Read, Write and Rewind make use of the two pointers in the header: the write pointer marks the end of the file, the read pointer points at the next byte to be read. For variable records, these pointers contain an offset, within the selected memory. For fixed records, they contain the record number.

Load

Loads a "program" file into memory (used by the Basic OLD statement, or by EA5 memory-image loaders).

Save

Saves a "program" file into the appropriate "ramdisk". If needed, it initialises the selected memory area, by installing the 8-byte header. It mercilessly overwrite any file currently in the "ramdisk", without any warning. EXPMEM1 suffers from the same bug than with OPEN when the Mini-memory is initialized.

Delete

Removes the >5AA5 flag and restores the loader pointers if appropriate.

Status

Returns informations about the file in a given memory area. The file type is taken from the 8-byte header.

For Fix files, the record number passed in the PAB is compared to the size of the memory area, and the "memory full" bit is set if needed. It is then compared to the write pointer in the header, to set the EOF bit.

For Var file, the EOF bit is set if the read and write pointers are identical.

The "not found" bit and the "write protected" bit are not used. If the memory area does not contain any file, the status byte will be >00.

```
Bit Value Meaning___________________
0   >80   Not used (file not found)
1   >40   Not used (write protected)
2   >20   Not used
3   >10   Internal (else Display)
4   >08   Program file
5   >04   Var records (else Fix)
6   >02   Memory full
7   >01   End of file
```

Checking memory usage

As we saw, there are two ways the Mini-memory can use the available RAM: one is to store files via the DSRs, the other is to load assembly language, via the LOAD subprogram. These two ways are not fully compatible, in that they tend to "fight" for memory, so if you want to mix them you will have to exercise some care. In particular, neither DSR will issue any warning before wiping out any assembly language program that you may have loaded into the memory expansion or into the cartridge RAM. Conversely, CALL INIT wipes out the file loaded into MINIMEM and makes it difficult to access the other two.

One usefull thing to do is to PEEK the first addresses of each domain, to know whether a Mini-memory DSR has placed a file in there. The addresses are:

- 28672 for MINIMEM (i.e. >7000)
- 8192 for EXPMEM1 (i.e. >2000)
- -24576 for EXPMEM2 (i.e.>A000)

And the Basic instructions would be something like this:

```
100 CALL PEEK(ADDR,FLAG1,FLAG2,TYPE,RECLEN,WR1,WR2,RD1,RD2)
110 WRITPT=(WR1*256)+WR2
120 READPT=(RD1*256)+RD2
```

FLAG1 should be 90 and **FLAG2** should be 165 if a file is present.
TYPE is 0 for Dis/Fix, 8 for Int/Fix, 16 for Dis/Var, 24 for Int/Var, and 255 for programs.
RECLEN is the record length (irrelevant for programs).
WRITEPT is the position of the write pointer, i.e. the current size of the file. For "program" and "variable" files the size is given in bytes. For "fixed" files it is a number of records, so you must multiply it with RECLEN

to obtain the number of bytes.
READPT is the current position of the read pointer, in the same units at WRITEPT.

If the Mini-memory was initialized (either from its menu, or from Basic with a CALL INIT), you will find 165 in FLAG1 and 90 in FLAG2 when PEEKing at 28672, rather than the other way around. If so, you absolutely cannot use the MINIMEM file since the Mini-memory RAM is needed by the loader to place its data, including the symbol table.

As for the memory expansion, you can check the loader's pointer at >7022 and >7026 with:

```
100 CALL PEEK(28706,A1,A2,B1,B2,C1,C2,D1,D2)
110 FSTHI=A1*256+A2
120 LSTHI=B1*256+B2
110 FSTLO=C1*256+C2
120 LSTLO=D1*256+D2
```

FSTHI is the first free address in the high memory expansion. If it's -24576, nothing was loaded yet.
LSTHI is the last address that can be used by the loader in the high memory expansion.
FSTLO is the first free address in the low memory expansion. If it is 8192 nothing was loaded yet.
LSTLO is the last address available for assembly language in the low memory expansion
If a couple of values reads as two zeros, it means that loading was disabled in the corresponding area, most probably because we are already using EXPMEM1 or EXPMEM2.

Incidently, you can get the same info for assembly programs loaded in the Minimemory RAM, by performing a similar CALL PEAK at >701C:

```
100 CALL PEEK(28700,A1,A2,B1,B2)
110 FSTMOD=A1*256+A2
120 LSTMOD=B1*256+B2
```

If you are interested in these issues, you can download MEMMODCK.TI, a Basic program by Paul Schippnick that is included here with his permission. This program reports on the memory in your system. It works with the Mini-memory module, but also with Editor-Assembler and Extended Basic. To transfer it to your TI-99/4A, see the instructions on my download page.

Using the DSRs with CALL INIT

Doing a CALL INIT with DSR files presents in memory will have catastrophic results for MINIMEM. The file will be completely wiped out, each and every byte zeroed even before the loader is installed. Note that the opposite is also true: MINIMEM overwrites the loader's private data and eventually any assembly program loaded in the Mini-memory RAM.

EXPMEM2 also seems to disappear, but the file is actually still there. What hapenned is that CALL INIT checks for the presence of the memory expansion by inverting the byte at >A000 and cannot be bothered to restore it afterwards. So we'll have to do it ourselves to recover the file:

```
CALL LOAD(-24576,90)
```

There is no such problem with accessing an existing file in EXPMEM1 after a CALL INIT.

However, it is extremely tricky to create new files after the Mini-memory has been initialized. The problem likely originated when TI programmers decided to integrate these DSRs and make them compatible with the CALL LOAD assembly loader. They wrote a quick hack to this effect and obviously never really tested it. Unfortunately, they managed to leave two very nasty bugs within 10 lines of GPL! As a result, EXPMEM1 and EXPMEM2 will not work properly when creating new files (with either SAVE or OPEN) after the Mini-memory was initialized. Although, if you really need to do this, there is a way around the bugs.

The DSR bugs

In an attempt to make the DSRs compatible with the assembly loader, TI programmers introduced two modifications:

- When creating a new file, SAVE and OPEN call a routine that installs the file header into the relevant memory area. This routine was modified so that it will also clear the loader's pointers for this memory area, so that no assembly will be placed in there.
- The DELETE opcode restores the loader's pointers to their original values.

Unfortunately, the first modification includes two very annoying bugs. If you are curious, have a look at the mmg.txt file: the bugs are highlighted with *** marks.

The first bug results from a stupid typo in a single byte, that ends up incrementing the wrong data word: >835A instead of >834A. This bug has two deleterious effects, wich only occur when the Mini-memory was initialized and you create a new EXPMEM1 file (with either SAVE or OPEN in output mode, or OPEN in update mode if the file does not already exist).

- Instead of preventing the loader from loading assembly in the low memory expansion, the bug actually disables loading in the HIGH memory expansion.
- The file header for EXPMEM1 is placed at >2004 instead of >2000. This is very unfortunate, because all other opcodes expect the header to lie at >2000 and will cause a file error when called under these conditions. As a result, we can create a file but not use it! Note however, that a file created before the CALL INIT can still be accessed normally.

The second bug was harder to catch. It results from the fact that the programmer loaded a word pointer in >834A, which will automatically overwrite >834B...that was supposed to contain the file type! As a result, any file created in EXPMEM1 or EXPMEM2 is registered as Dis/Fix. Which means that, once you have closed the file, you cannot open it again unless it was indeed a Dis/Fix file. There is no such problem with SAVE however, because SAVE does not get the file type from >834B.

The best way to overcome these bugs is to trick the program into thinking that the Mini-memory was not initialized, so the buggy routine will not be called. This means that we will have to handle the loader's pointers ourselves, but that's no big deal. A simple way to do this is to clear the flag at >7000:

```
100 CALL LOAD(28672,0)
110 OPEN #1:"EXPMEM1",whatever
120 CALL LOAD(28672,165)
130 CALL LOAD(28710,0,0,0,0)
```

Line 100 clears the "initialized" flag and line 120 restores it. Thie latter is required because CALL LINK checks for this flag and returns an error if it did not find it. Note that there is nothing wrong with delaying this restauration until just before the CALL LINK.

Line 130 clears the loader's pointers for the low memory expansion (at >7026), which EXPMEM1 will be using. This is required so that further CALL LOAD("filename") statements will not load assembly language programs into the low memory expansion. For EXPMEM2, you would use address 28706 instead. You only need to do this once for EXPMEM1 and once for EXPMEM2. Unless you subsequently perform a DELETE on either EXPMEM1 or EXPMEM2, in which case the corresponding loader pointers will be automatically restored. If you change your mind and want to open or save a new file after a delete, you will again need to clear the corresponding loader's pointers.

A more sophisticated way would be to reserve some room for your files and tell the loader to place assembly afterwards. This can be done be loading a value between >A000 and >FFE0 (i.e 160,0 and 255,240) into 28706 and a value between >2000 and >3FFE (i.e. 32,0 ad 63,254) into 28710. You could either choose an arbitrary value for this border or fetch the current size of the DSR file (see `WRITEPT` in the previous section) and use it to place assembly just after the end of the file. All this is a bit dangerous though, because nothing will prevent you from overwriting assembly by increasing the size of the file in EXPMEM1 or EXPMEM2. You just have to know what you are doing...

Revision 1. 7/18/99 Ok to release

Revision 2. 7/12/01 Correctedtypos, CRU address, added memory checks, bugs discussion.
Revision 3. 8/26/01 Added link to picture page.

Back to the TI-99/4A Tech Pages