GreenArrays®

Application Note AN009 *Revised 9/12/12*

Attaching a PS/2 Keyboard

This document describes a method for attaching a PS/2 keyboard to the EVB001 evaluation board. The whole process from the hardware to arrayFORTH till polyFORTH is covered by this app note.

The text assumes you have familiarized yourself with our hardware and software technology by reading our other documents on those topics. The current editions of all GreenArrays documents, including this one, may be found on our website at http://www.greenarraychips.com . It is always advisable to ensure that you are using the latest documents before starting work

The work described herein is that of Stefan Mauerhofer.

Contents

1. Introduction

This application describes a way to attach a PS/2 keyboard to the EVB001 evaluation board using a small interface for level shifting between 1.8 and 5 volt circuitry. 4 nodes are used to drive the interface and sending/receiving bytes. The more complex protocol for sending bytes and waiting for acknowledgement and interpreting the scan codes is handled in the polyFORTH virtual machine.

2. The PS/2 Interface

PS/2 is a standard for attaching devices, mostly input devices like keyboard or mouse, to a computer. A good description of the line operations can be found in [1].

2.1 Physical Attachment

The PS/2 interfaces can be identified by the 6-pin mini-DIN socket. It uses a serial 2-wire bidirectional protocol for transmitting bytes.

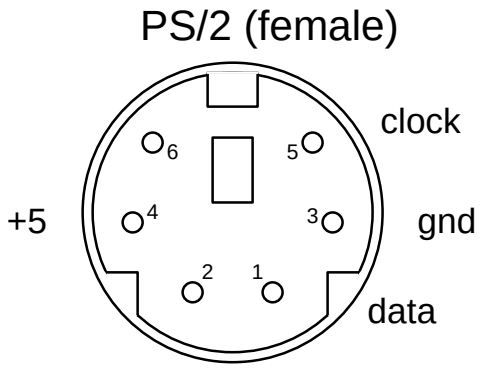


Fig. 1 Mini-DIN PS/2 connector

Both lines (clock and data) are open-collector. That means they are low active and must be connected to 5V with a pull-up resistor, which allows communication in both directions.

The PS/2 interface implements a bidirectional synchronous serial protocol. The bus is “idle” when both lines are high (open-collector or open-drain). This is the only state where the keyboard/mouse is allowed begin transmitting data. The host has ultimate control over the bus and may inhibit communication at any time by pulling the clock line low.

2.2 Interface Circuit

The circuit for attaching a PS/2 device is quite simple. A TXS0108E open-drain level-shifter chip must be connected, because the PS/2 interface operates at 5 volts. Figure 2 shows the circuit:

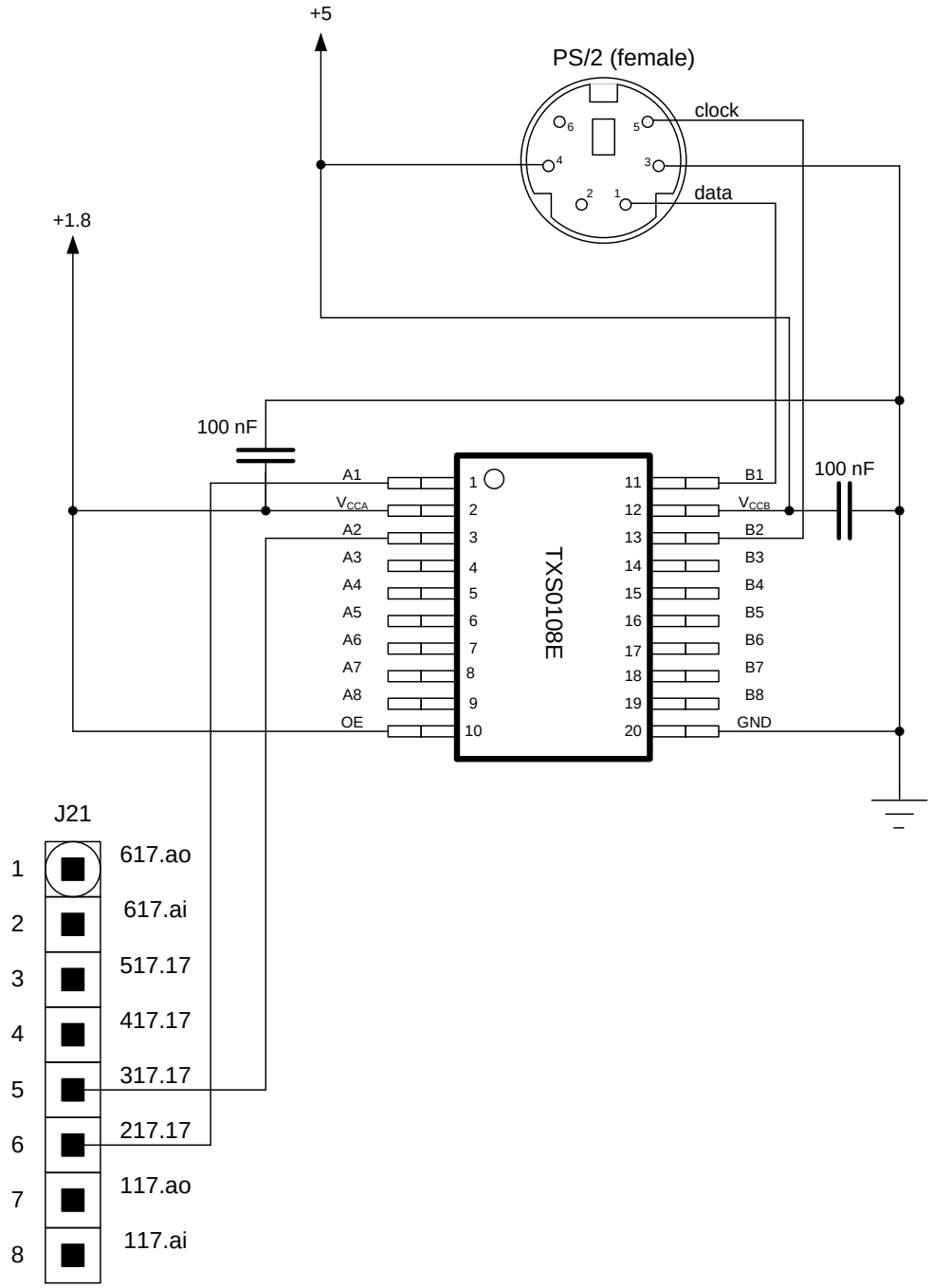
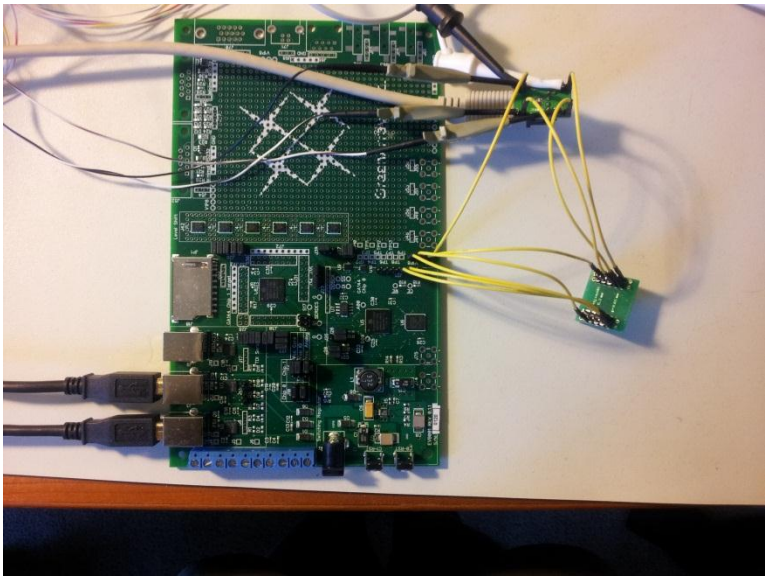


Fig. 2 Interface circuit

No pull-up resistor is necessary, because it is already built into the chip. For details please consult the datasheet from Texas Instruments. The two decoupling capacitors can be neglected for a test environment, because all frequencies are below 20 kHz, which lies way below the limits of the level-shifter chip and the GA144. Picture 1 displays the experimental circuitry used for this application note.



Pic. 1 Test circuit

Picture 1 displays the test buildup. The level-shifter chip works fine in this configuration even without the 2 decoupling capacitors, although I recommend using them.

2.3 General Mode of Operation

When the interface is inactive, then both lines are at 5V. Normally the lines are controlled by the keyboard and the clock frequency ranges from 10 to 18 kHz. Figure 3 shows the timing, when the keyboard is sending a byte:

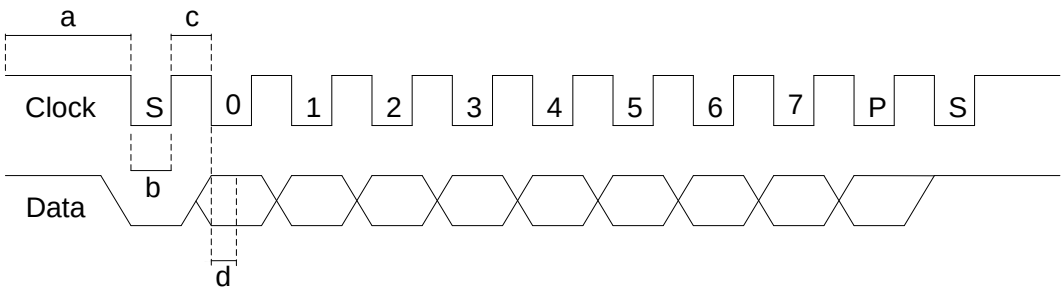


Fig. 3 Read operation and general timing

The bit sequence is: start, 8 x data (least significant bit first), parity (odd), stop.

The timing limits are:

a > 50µs

30µs < b, c < 50µs

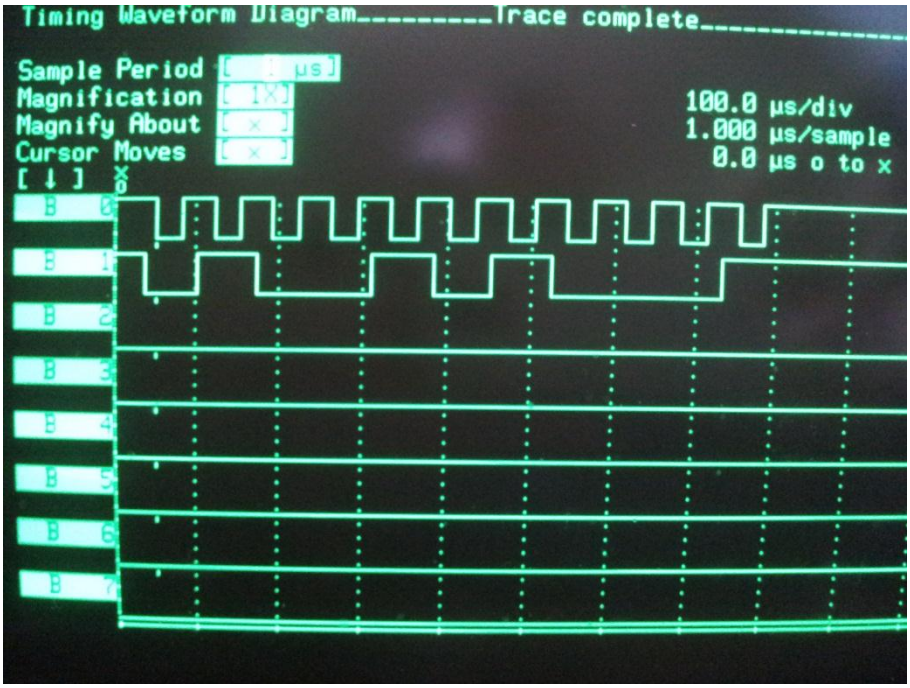
d ≈ 10-20µs.

The keyboard controls both lines, so the GA144 should set the corresponding pins to high impedance input. Initially both lines (Data and Clock) are pulled to +5V (high).

Note: The clock pulses (for read and write!) are always generated by the keyboard. This makes integration into a GA144 easier, because no timer is needed.

2.3.1 Receiving Data

If the keyboard wants to send a byte it sets the data line low (start bit) followed by 8 data bits (LSB first), parity bit (odd) and a stop bit (high). The data line is changed during the high phase of the clock and stable during low phase. Picture 2 displays a screenshot from a logic analyzer's display while reading the byte 29h from the keyboard.



Pic. 2 Logic analyzer display while reading a byte from the keyboard

2.3.2 Sending Data

Because the lines are open-collector, it is possible to send a byte to the keyboard. The timing for this case is shown in figure 4:

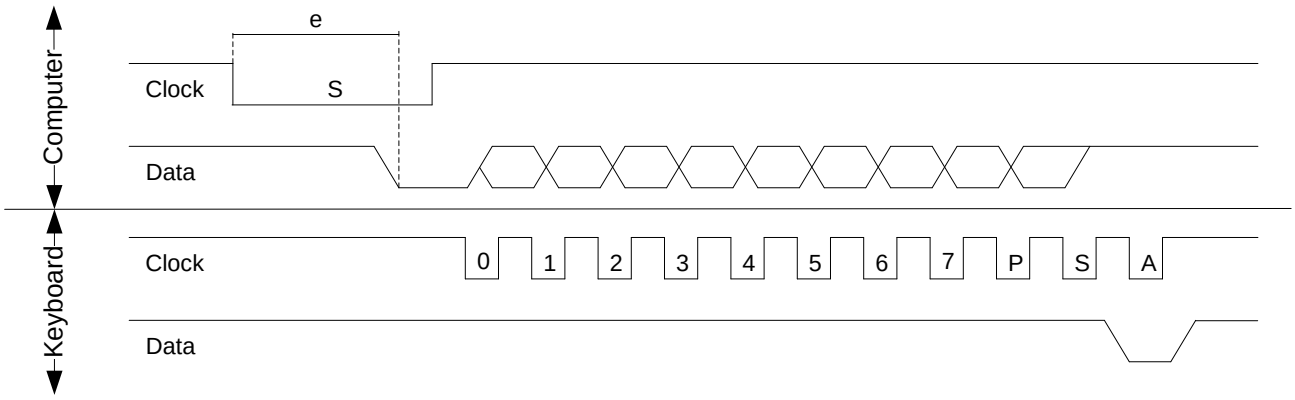
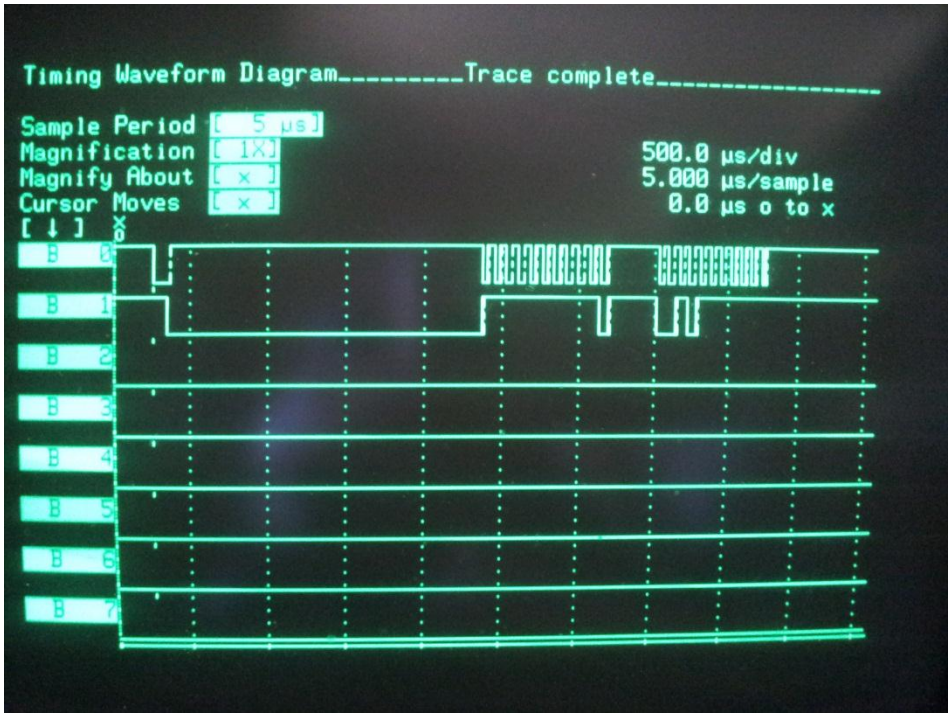


Fig. 4 Write timing

Sending data to the keyboard is little more complicated than receiving. First the host sets clock low for at least 100 microseconds (100 µs < e < 10 ms). Then it sets the data line low, signaling that it wants to send data. Now the clock line is released. The host waits until the keyboard takes over and begins clocking. The data line is altered by the host during low clock phases and it holds the data while the clock is high. Note that is the inverse behavior compared to reading. The host releases the data line when the stop bit is reached. The keyboard then sends an acknowledgment bit for finishing the operation. Picture 3 displays the logic analyzer display of a keyboard reset sequence where FFh (reset) is sent to the keyboard that responds with FAh (ack).



Pic. 3 Sending reset (FFh) to keyboard with response (FAh)

Note the relative large delay (2 ms) from the clock release till the keyboard starts clocking. This is not a problem since the GA144 does not measure any time but waits for events to occur.

3. arrayFORTH Implementation

4 nodes are used to implement the basic serial protocol in F18 code. Fortunately the keyboard does generate the clock signal for reading and writing, so the nodes need no timed mechanism for generating a clock. Node 317 is used for the clock line and node 217 for the data line. Node 316 is a 1 entry sending buffer and node 216 a receiving 32 element FIFO buffer.

3.1 The Data Line

The data line is handled by node 217. Listing 1 is the code for this node:

```
ps/2 interface driver - data,
connect data to pin 17 @ node 217
buffer bits to send or receive
bit current bit mask
start high and jump to down port
clear write n @ 0 and 1 @ 1
bo write next bit
5us wait approx. 5 microseconds
high put data line to high impedance
low drive data line to ground
bi read next bit
r1 read a 1 bit
next-bit set bit mask to next bit
parity nondestructive calculation of even parity
with output 0 or hex 100
byte-out read 1 byte from node 317 and calculates
odd parity and stop bit
accept finish reading of a byte
error parity error detected
ok pass received byte to next node
ack read acknowledge bit
```

```
920 list
ps/2 interface driver - data,
reclaim 217 node 0 org
buffer 00 0 ,*bit 01 0 ,
start 02 leap -d-- ;
clear 04 dup or dup a! !+ a ! ;
bo 07
5us 07 2000 for unext,
dup or a! @+ @ dup 2* ! and if
high 0d swap then dup or !b ; then
low 0e 20000 !b ;
bi 10 dup or a! @b -if
r1 12 @+ @ or dup dup or a! !,
then*next-bit 15 @+ @ 2* ! ;
parity 17 100 a! dup dup or over 7 for pn,
2* dup a and push over pop or over next drop ;
byte-out 21 down a! @ dup push parity,
pop or dup dup or a! 300 or !+ a ! ;
accept 29 dup or a! @ parity or 100 and if
error 2e ; then
ok 2e dup or a! @ ff and right a! ! ;
ack 33 high ;
```

Listing 1 Data line

After starting the node waits on the down port for code from node 317. The port executes directly any cell received from the clock node 317. Node 217 is offering a set of entry points that can be called by node 317.

3.2 The Clock Line

Node 317 is responsible for the clock line. It needs to place call instructions into the down port. The sequence **`@p .. xxx ..`** pushed the call instruction onto the stack. Since we didn't use **`reclaim`** we still have the word addresses compiled from block 920 and we can use them for block 922.

```
ps/2 interface driver - clock,
,
connect clock to pin 17 @ node 317
wait0 wait until clock line is 0
wait1 wait until clock line is 1
20us wait approx. 20 microseconds,
this delay is critical for correct operation
waitr wait for the clock going 1 and then 0
da set port a to clock node
from-ps/2 read a byte from ps/2
stopbit read and handling the stop bit
to-ps/2 write a byte to ps/2
start entry point of node
loop poll the clock line or the port write bit
to either begin a reading or a writing operation .
this loop runs indefinitely leaving the node
running .
```

```
922 list
ps/2 interface driver - clock,
317 node 8 org
wait1 08 dup dup or ahead
wait0 0a 800 then 0c !b left a! @ drop
da 0f down a! ;
waitr 11 wait1 leap wait0 then
20us 14 8000 9000 10000 for unext ;
from-ps/2 17 da @p .. clear .. ! 20us 1b,
9 for waitr @p .. bi .. ! next,
stopbit 21 wait1 @p .. accept .. ! 20us ;
to-ps/2 25 right a! @ 20000 !b da,
@p .. byte-out .. ! ! 60000 for unext,
startbit @p .. low .. ! 4000 for unext,
dup or !b wait1 9 for 20us wait0,
@p .. bo .. ! 20us next waitr 3d,
@p .. ack .. ! waitr ; 40 0 org
start 00
loop 00 @b - -if from-ps/2 loop ; then 03,
- 8000 and .. 05 if to-ps/2 then loop ; 08
```

Listing 2 Clock line

The memory of the node is quite full. The main loop is spinning (thus burning around 4 mA) and waits either for a clock low pulse from the keyboard or an input from the right port. Once an operation (either sending or receiving) has started it will be completed without possibility of interruption. Since there are no timeouts used in the design it is not possible for the node to resynchronize its operation with the keyboard once a single bit has been added or is missing. Therefore the whole interface is not very robust regarding bit synchronization errors. However the stability is sufficient for an experimental environment.

There are 3 constants (10000, 60000 and 4000) that are important for the timing. It turns out that the timing parameter in the word **`20us`** is crucial. Once the node does lose a bit, its behavior is undefined and the chip must be reset. My keyboard is working without problem with values 9000 or 10000 but runs into sporadic error when using 8000.

Node 217 (data) and 317 (clock) work together very closely. This is an example of how to use the port execution facility of the GA144 in an efficient way.

3.3 The Data Buffer

Incoming data is buffered in a FIFO buffer in node 216. My first implementation was designed to block and wait for input from nodes 215 and 217 (left and right):

```
ps/2 interface driver - data buffer,
implementing a ringbuffer for 31 cells.
r! store read pointer
r@ fetch read pointer
w! store write pointer
w@ fetch write pointer
inc increment pointer and clip to buffer
-empty? test if buffer is not empty
-full? test if buffer is not full
start begin of program
loop processing loop . wait for command or data .
for a node to read a negative word must be send
read read from buffer
write write to buffer
```

```
924 list
ps/2 interface driver - data buffer,
reclaim 216 node 20 org
r! 20 @p drop !p ;*r@ 21 0 ;
w! 23 @p drop !p ;*w@ 24 0 ;
inc 26 n-n 1 . + 1f and ;
-empty? 2a -f w@*check? r@ or ;
-full? 2d -f w@ inc check? ;
start 30*loop 30 @b -if
read 31 -empty? if r@ dup inc r! a! @ dup then
drop !b loop ; then
write 39 -full? if drop w@ dup inc w! a! ! then loop
; 40
```

Listing 3 FIFO buffer (flawed design – not used)

There is a problem with this design. Both nodes 215 and 217 act totally independent from each other, thus it is possible that a node starts writing while the data from the other node is currently reading. In this case node 216 may read garbage and the behavior is undefined. The only way to solve this is to poll both sides continuously and to sacrifice some energy for keeping the node running. Greg Bailey gave this info to me as I was writing this app node and he reviewed my solution. This problem may be fixed in a future version of the chip.

```
ps/2 interface driver - data buffer,
implementing a ringbuffer for 15 cells.
r! store read pointer
r@ fetch read pointer
w! store write pointer
w@ fetch write pointer
inc increment pointer and clip to buffer
-empty? test if buffer is not empty
-full? test if buffer is not full
start begin of program
loop polling loop
```

```
926 list
ps/2 interface driver - data buffer,
reclaim 216 node 10 org
r! 10 @p drop !p ;*r@ 11 0 ;
w! 13 @p drop !p ;*w@ 14 0 15 ;
inc 16 n-n 1 . + f and ;
-empty? 1a -f w@*check? r@ or ;
-full? 1d -f w@ inc check? ;
start 20 dup !b drop
loop 21 a! @b and or if drop @ push -full? if
drop w@ a! pop ! w@ inc w! dup then then drop
drop,
a! @b and or .. if drop @ push -empty? if drop
r@ a! @ push r@ inc r! dup then drop a! pop !
loop ; then drop drop loop ; 3c
```

Listing 4 Fixed FIFO buffer using poll loop

Block 926 contains the looping variant, which works correctly in all circumstances. The preset circular stack (initialized in block 912) is used to feed the loop with the data it needs to function thus saving precious memory on the node leaving enough space for 16 values in the FIFO buffer.

3.4 Send Buffer

Node 316 contains a simple wire node for decoupling a polyFORTH write to keyboard from the current operation. In the case that the keyboard is sending a character, node 317 does not accept any data. A wire node is also always a 1-element Buffer. In this sense node 316 is used to prevent any writer from blocking.

```
928 list
ps/2 interface driver - decoupling wire node,
reclaim 316 node 0 org
start 00 dup or -,
dup push dup push dup push dup push,
dup push dup push dup push dup push,
for begin @ !b unext unext 07
```

Listing 5 Wire node

3.5 Loading

A loading block is used to load all 4 nodes. Block 200 must contain the instruction `918 load`.

```
918 list
ps/2 loader,
reclaim,
920 load 922 load 926 load 928 load,
,
reclaim
```

Listing 6 Loading block

3.6 Initialization

Block 912 is used for node initialization. This can be used for softsim or loading the code into the GA144 chip or integrate it into the polyFORTH boot stream.

```
912 list
ps/2 - code loader,
217 dup +node /ram 2 /p io /b,
317 dup +node /ram 0 /p io /b,
216 dup +node /ram 30 /p left right or 155 or /b,
316 dup +node /ram 0 /p left /a right /b,
```

Listing 7 Initialization block (early version - not used)

`/p` defines the starting point of the code. The expression `left right or 155 or /b` requires a little explanation. The goal is to initialize the register **b** with the combination of port **left** and **right**. Inside the F18-compiler we could use the predefined address **r-l-**. Here we don't have these words but we can use the port words. We can combine any ports by x-oring the port addresses. Since port addresses are x-ored with 155h we must add this operation if the number of port is even. We use 2 ports therefore we must x-or the result with 155h to get a correct port number.

This was my initial design. Since we cannot use the blocking code on node 216, we must use the polling code, which halves the available buffers cells from 31 to 15. But 16 cells are still enough for our purposes. The new code requests the stack to be filled with predefined values for saving node's memory and use the cycling stack. Listing 8 shows the modified initialization code.

```
912 list
ps/2 - code loader,
217 dup +node /ram 2 /p io /b,
317 dup +node /ram 0 /p io /b,
216 dup +node /ram 20 /p io /b left 0 800 left right
0 8000 right 201ff 9 /stack,
316 dup +node /ram 0 /p left /a right /b,
```

Listing 8 Initialization block with poll loop fix and stack prefill

3.7 Receiving Data

This and the next chapter explains how the clock node invokes calls to the data node and when.

First Node 317 waits for the clock line to go low. If this happens it starts a read cycle by calling **from-ps/2**. Figure 5 shows what happens in this word:

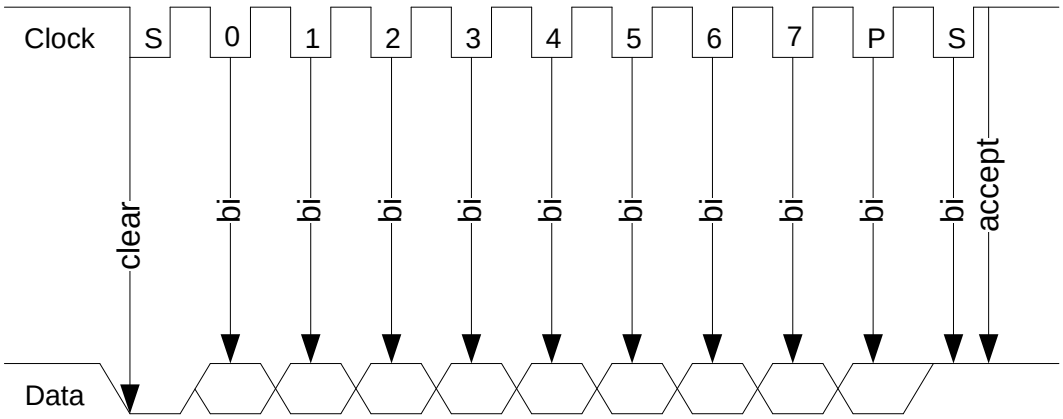


Fig. 5 Node interaction for receiving a byte from the keyboard

The sequence **clear**, 10 x **bi** and **accept** is called from node 317 via the down port, timed by the clock line. Accept retrieves the byte and sends it to node 216. There it is buffered for later use e.g. by polyFORTH. Up to 15 bytes can be buffered in node 216 before it overflows and ignores any further data. To read a byte from node 216 any word must be sent to it from node 215. If an unread byte is available in the FIFO buffer, then it is returned otherwise the word sent from node 215 is returned, signaling a buffer empty condition.

3.8 Sending Data

Node 317 initiates sending data if it has been idle.

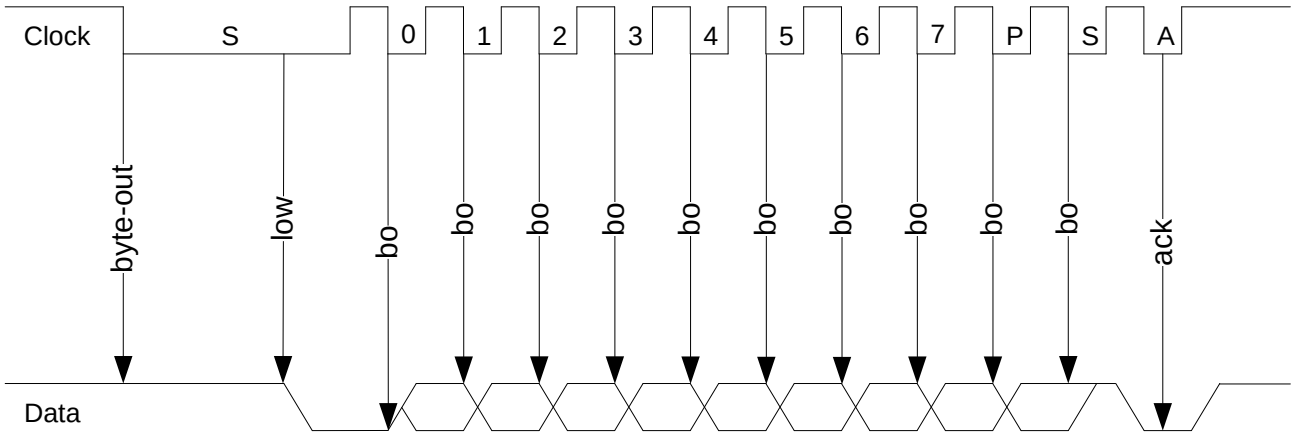


Fig. 6 Node interaction for sending a byte to the keyboard

The sequence `byte-out` followed by the data, `low`, 10 x `bo` and `ack` is called from node 317 via the down port, timed by the clock line. The clock line must be driven to low long enough that the keyboard does recognize a sent condition or aborts any sending operation. This waiting is done by `100us` in node 317. Node 217 calculates also the parity bit and sets the stop bit.

3.9 SoftSim Integration

If we want to test our interface with SoftSim then block 916 is used to define the environment.

```
916 list
ps/2 - softsim starter,
912 load
/clk softbed assign time @ 100 / 1 and 1 or ?v
p17v ! ;,
317 !node /clk,
/dat softbed assign time @ 1300 / 1 and 1 or ?
v p17v ! ;,
217 !node /dat,
```

Listing 9 SoftSim integration

A `916 load` must be placed in block 216 to include our example in SoftSim. The level at the clock line is changed every 100 ticks and on the data line every 1300 ticks. In order to see something we must change 3 timing constants in block 920 and 922, e.g. replace 40000 and with 4. After using SoftSim and before using the code on the chip we must undo all changes.

3.10 Testing on the Chip

For testing on the host chip we must have a block that loads the code onto the chip and another for retrieving the scan codes.

```
914 list
ps/2 - loader,
empty compile host load loader load,
0 708 hook 0 -hook,
912 load,
2 ship
```

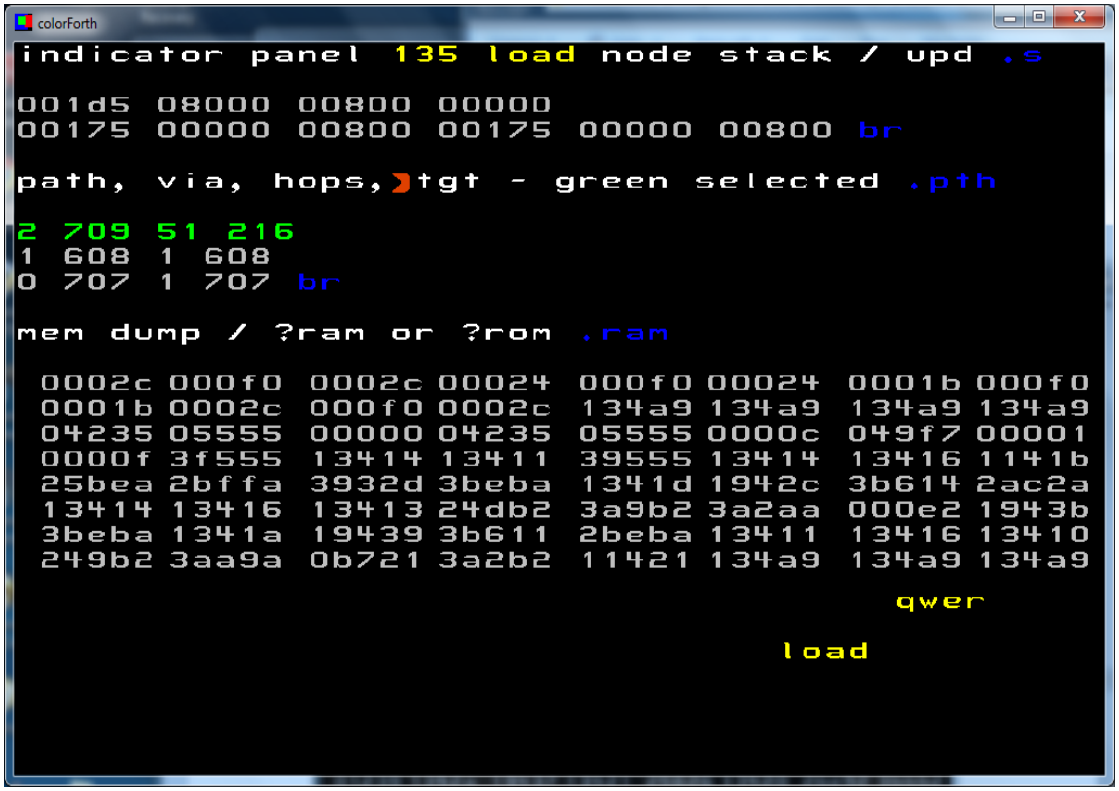
Listing 10 Loading code onto the chip

First we type `914 load`. After some time the code is loaded into the chip and started. We can now type something into the keyboard, e.g. “test”. Now we must use `910 load` to reset the chip and display the FIFO buffer.

```
910 list
ps/2 - loader,
empty compile host load talk,
2 216 hook panel ?ram
```

Listing 11 Resetting the chip and displaying the FIFO buffer node

We should now see the following screen:



Pic. 4 Screenshot after typing “test” on the keyboard

We can now see the make- and break-codes the keyboard sends for every keystroke. **2c** is the make code for the key **t** and **f0 2c** is the break code for the same key. After reset most keyboards use the scan code set 2. See [2] for details.

4. polyFORTH Implementation

For using the full potential of the PS/2 interface, we can integrate it into a virtual machine that takes care of the higher level aspects. The larger memory of the polyFORTH virtual machine is better suited for configuring the keyboard and interpreting the scan codes.

4.1 Integrating the Keyboard Code

If we want to load our PS/2 code together with the polyFORTH virtual machine, then we can place a `912 load` into block 368 (or 478 for older versions), where the additional I/O for the virtual machine is loaded:

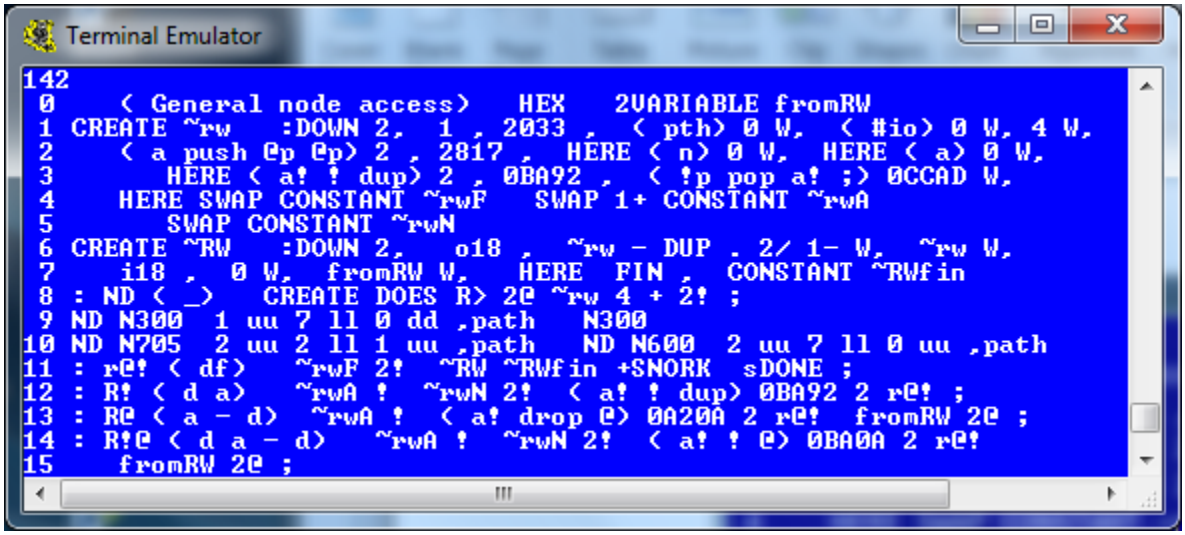
```
368 or 476 list
- additional i/o,
spi 705 +node 1606 /ram io /b a9 /p,
,
keybrd 912 load
```

Listing 12 Integrating our code into the polyFORTH boot stream

When polyFORTH is started then our code will also be loaded into nodes 216, 217, 316 and 317. That is true whether you start polyFORTH from the IDE (**`450 load`**) or you install the polyFORTH boot stream in the flash (**`460 load`**).

4.2 Accessing the PS/2 nodes from polyFORTH

After starting polyFORTH on the eval-board we must load the snorkel and ganglia mechanism to be able to access the keyboard nodes. **142 load** will do that. But before we load block 142 we must add the word **R!@** to it, if it is not already there (arrayFORTH version prior to h). Listing 13 displays the block 142 for arrayFORTH versions h or higher.

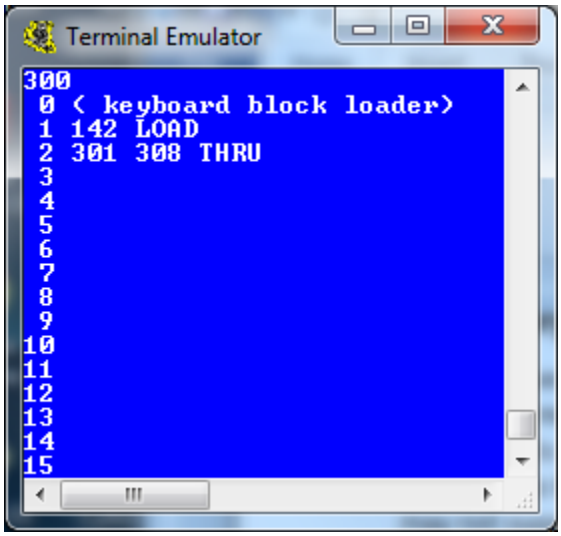


Listing 13 Block 142 (arrayFORTH version h and later)

4.3 Integration into polyFORTH

Until now we have used our keyboard interface “by hand”. In order to fully support the keyboard and its features we must implement a little driver that converts the scan codes into ASCII code and keep track of meta-keys like shift and caps-lock. To simplify the task we assume that our keyboard supports the scan code set 3 (for details see [3]). Scan code set 3 is the simplest and most orthogonal set of the 3 scan code sets available. Be aware that some keyboards may not support scan code set 3. They will not work properly with the code provided in this app node.

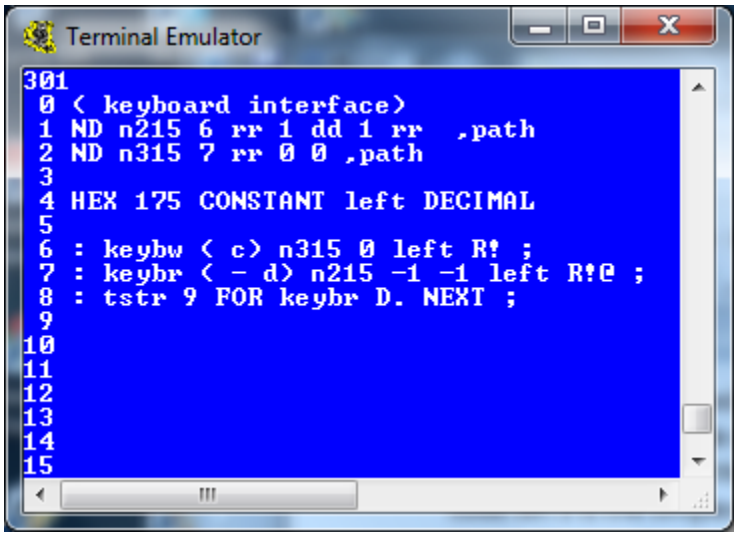
There is a block (300) for loading the whole keyboard driver.



Listing 14 Keyboard driver loading

4.3.1 Accessing GA144 Nodes

Block 301 defines the words (**keybr** and **keybw**) for actually accessing the PS/2 read and write nodes.



Listing 15 Accessing PS/2 nodes

Users of arrayFORTH version g or earlier must be aware that the snorkel/ganglia node is different from version h and higher (node 108 if version <= g else node 207). The path used in Listing 15 is for arrayFORTH version h or higher.

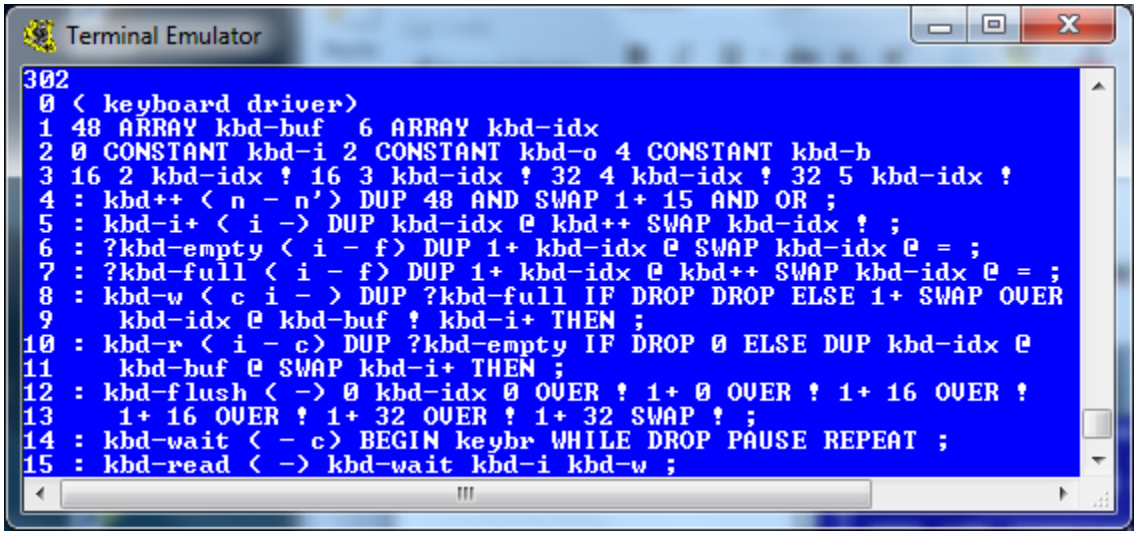
keybw writes a byte to the left port of node 315, thus writing to node 316. Node 316 is a wire node and reads the byte, if it is empty. Node 316 is basically not necessary but it decouples the polyFORTH virtual machine from node 317, which might be busy.

keybr reads the 15 byte FIFO buffer in node 216, accessing it via node 215. If the FIFO is empty then **keybr** returns **-1 3** otherwise the next byte is removed from the FIFO and **b @** is returned.

The word **tstr** is used to read and display 10 values from the buffer in node 216. It is convenient for testing.

4.3.2 Buffers and Meta-Key Handling

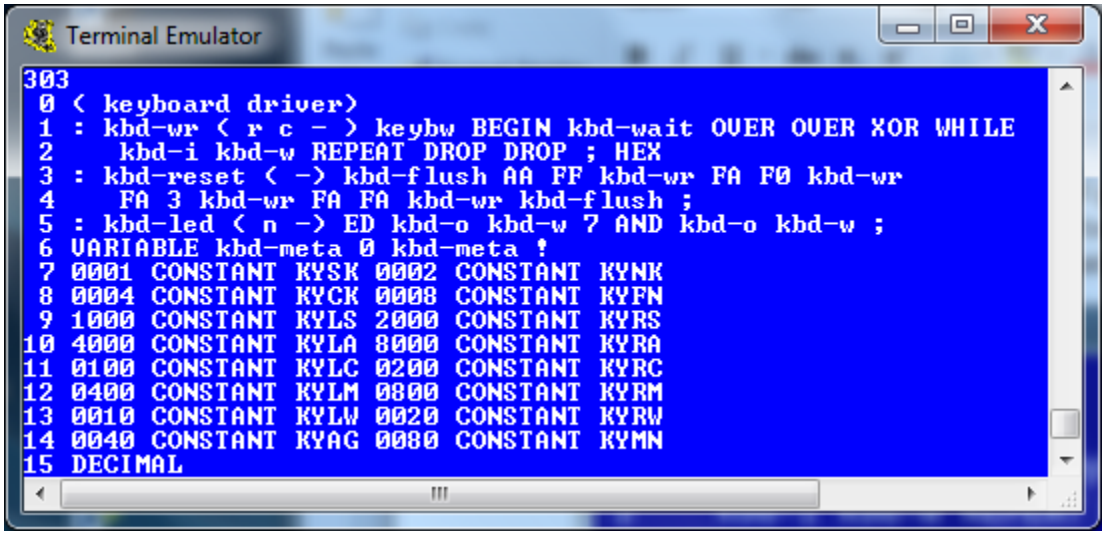
Block 302 contains words for handling 3 buffers: **kbd-i**, **kbd-o** and **kbd-b**. Each buffer contains 16 cells organized as a FIFO queue. An array with 6 elements contains the read and write indexes for the buffers. Buffers **-i** and **-o** are the raw input and output buffers, while buffer **-b** holds the ASCII keys after scan code translation.



Listing 16 Buffer and buffer management words

The word **kbd++** increments an index and limits it to the buffer it points to. An index is incremented with the word **kbd-i+**. The words **?kbd-empty** and **?kbd-full** are used for testing whether a buffer is empty or full. **kbd-w** writes a cell into a buffer, .e.g. **2 kbd-i kbd-w** writes 2 into the raw input buffer. **kbd-r** reads the next cell from a buffer or returns 0 if the buffer was empty. The word **kbd-flush** clears all 3 buffers. **kbd-wait** waits for the next byte from the PS/2 interface and **kbd-read** puts the next byte from the PS/2 interface into the raw input buffer.

A keyboard contains certain keys that do not generate an ASCII code but have an influence on the interpretation of other keys, e.g. shift or caps-lock. These meta-keys must be specially treated. Block 303 contains the constant definitions for these meta-keys and the variable **kbd-meta**, holding the state of the meta-keys.

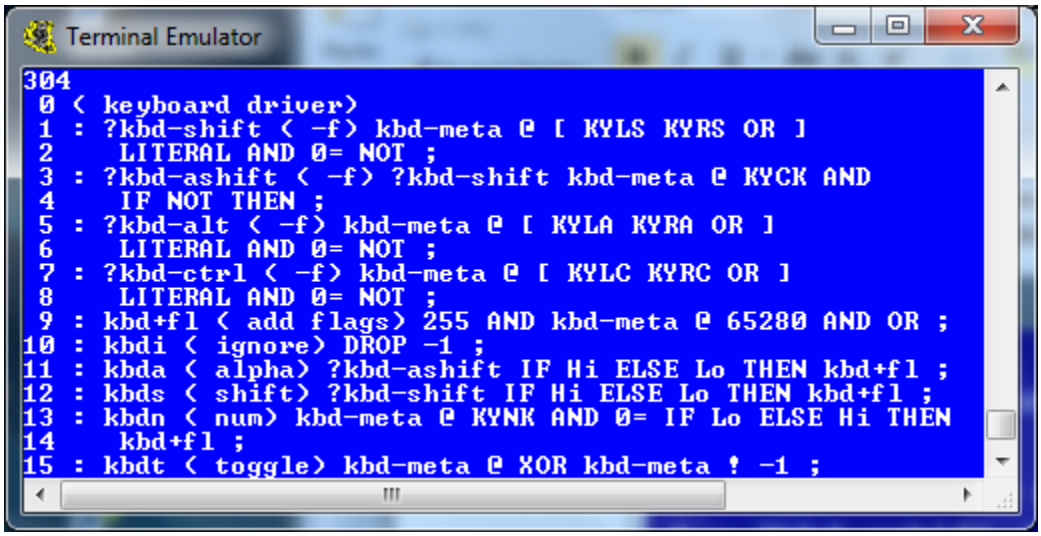


Listing 17 Meta key constants

Apart from the meta-key definitions 3 useful words are defined. First **kbd-wr** is used to send a byte to the PS/2 interface and to wait for a specific reply. The word **kbd-reset** resets the keyboard and configure it to use scan code set 3. With **kbd-led** the leds on the keyboard can be set. The state of a meta-key is defined by a bit in the variable **kbd-meta**. The bit numbers are defined so that the lower 3 bits can be directly used for setting the keyboard leds.

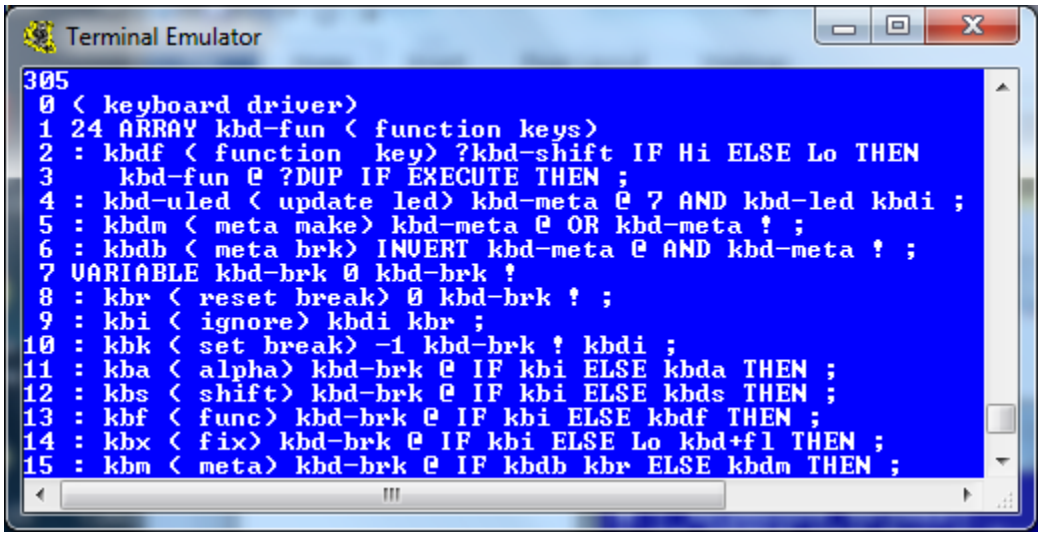
4.3.3 Key Types and Keymap Table

All keys can be grouped to key types with similar behavior, e.g. all alpha keys are sensitive to the state of the caps-lock and the shift-keys. Blocks 304 to 306 contain the words to define all needed key types.



Listing 18 Meta key words

The words **?kbd-shift** and **?kbd-ashift** are used for determining whether the basic value or the shifted value of a key must be taken. **kbd+fl** adds some meta-key flags to a character byte. **kbdi** is used when a key must be ignored. Most break codes can be ignored. **kbda** is used for alpha keys, **kbds** for shifted. Key on the numerical keybad must use **kbdn** and meta-key toggles (e.g. caps-lock) use **kbdt**.

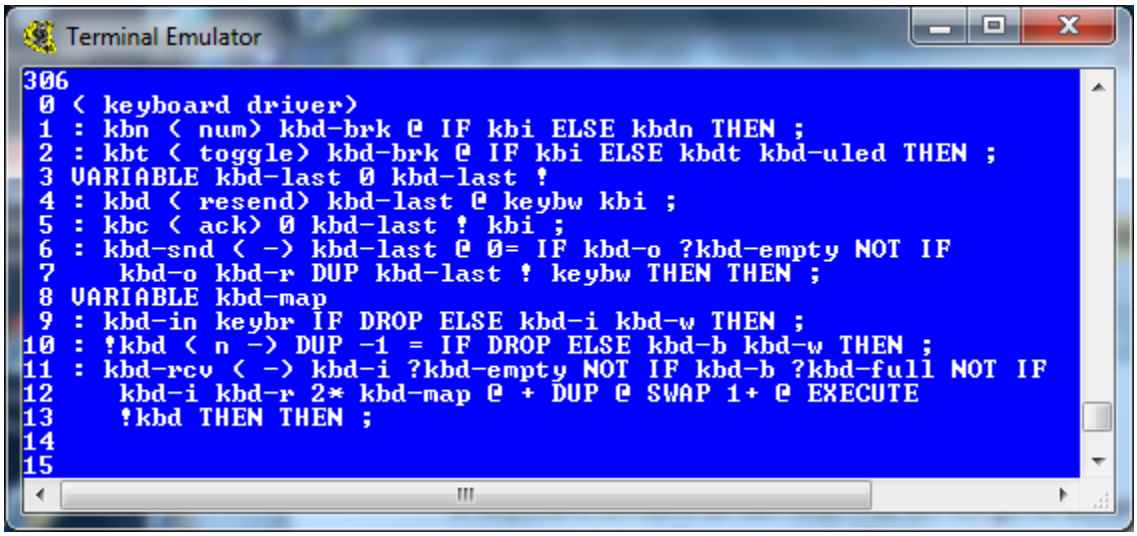


Listing 19 Function keys and break code flag

kbd-fun is an array of function key vectors. Index 0 to 11 are used for the function keys F1 to F12 while index 12 to 23 for the shifted function keys.

The word **kbdf** is used for function key, **kbd-uled** updates the keyboard leds and **kbdm** is used for meta-keys like shift. The variable **kbd-brk** holds a flag that is set when a break code (F0) is received from the keyboard, which informs the driver to ignore the next scan code.

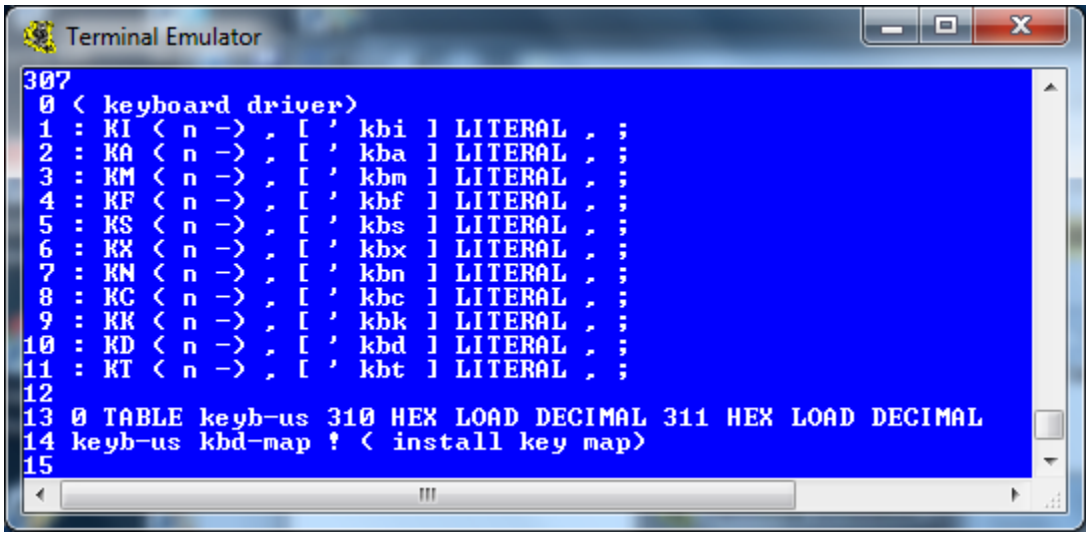
The 3 character words **kb_** handle the corresponding make and break codes.



Listing 20 Last byte send and keyboard processing words

The variable **kbd-last** holds the last byte sent to the keyboard. If it is not 0 then an acknowledgment from the keyboard is pending. It is also used to resend the byte when the keyboard replies with FE. The variable **kbd-map** contains the address of the current key translation table. kbd-snd and kbd-rcv are used to process the raw input and output buffers.

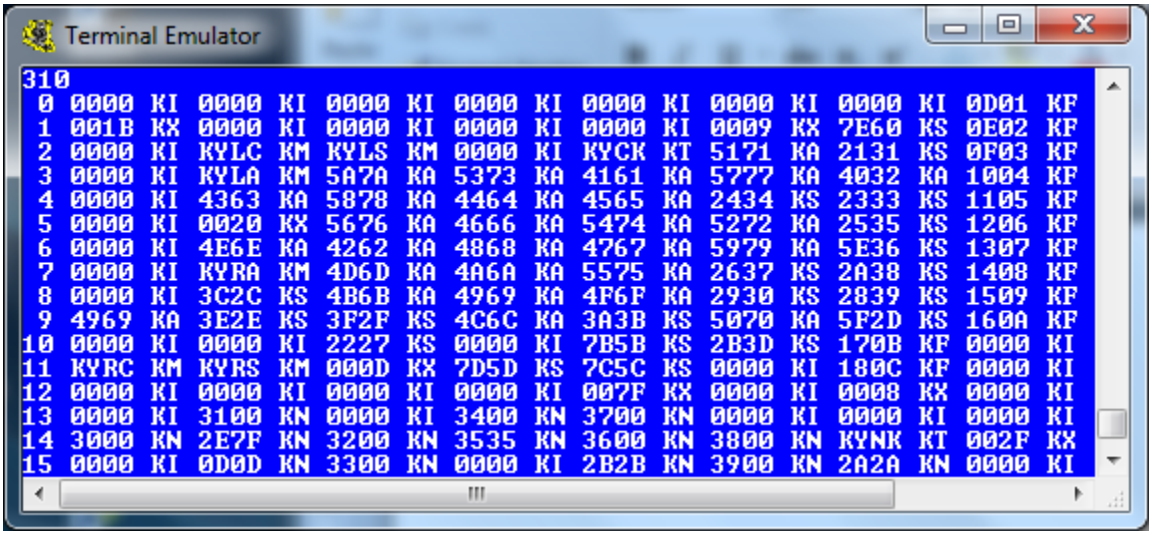
Block 307 contains the final 2 character words for compiling the key definitions.



Listing 21 2 character key compiling words

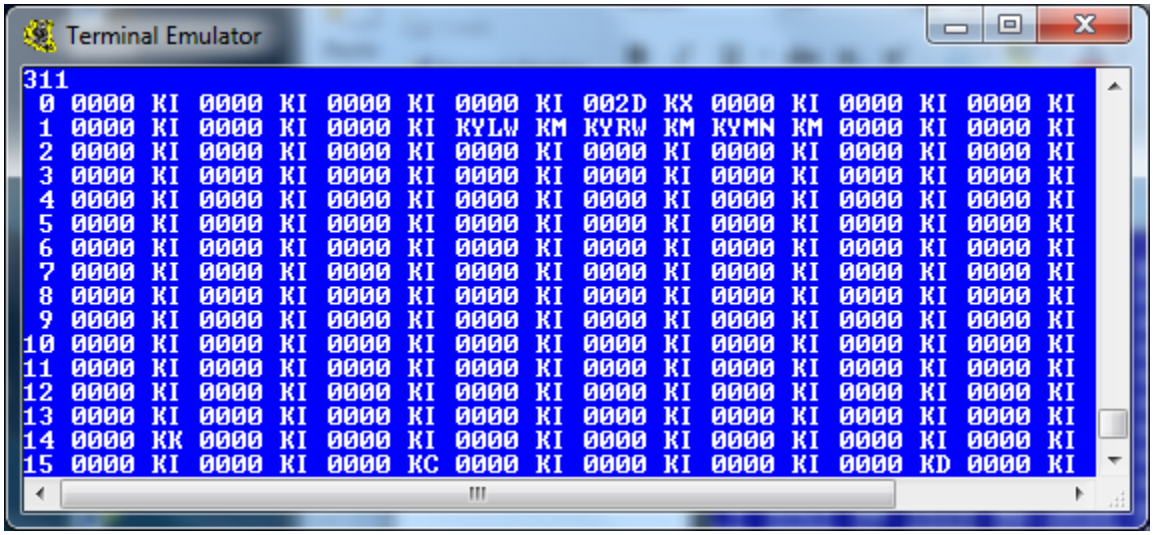
These 2 character short words are used to define the keys in the key map, which is loaded from block 310 and 311. The key types handled are **i**gnore (KI), **a**lpha (KA), **m**eta (KM), **f**unction (KF), **s**hift (KS), fi**x** (KX), **n**umeric (KN), a**c**knowledge (KC), resen**d** (KD) and **t**oggle (KT).

Block 310 contains the first half of the key map. It is used by **kbd-rcv** to convert scan codes to ASCII codes or to handle special keys. Even the handling of an acknowledgment (FA) or resend (FE) code from the keyboard is defined by this table.



Listing 22 US-keyboard map for scan codes 00 .. 7F

Block 311 contains the upper half of the key map.

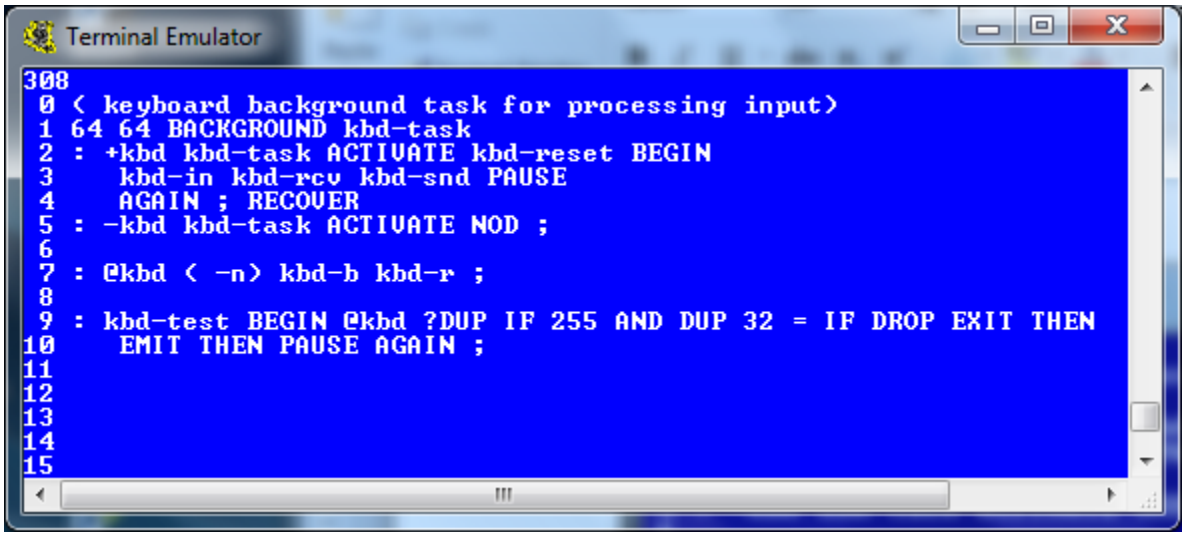


Listing 23 US-keyboard map for scan codes 80 .. FF

The key map used in this document is for a US-layout keyboard. For keyboards with other layouts the key map must be adapted and the keyboard driver eventually adapted.

4.4 Using a polyFORTH Background Task

Finally we can put all pieces of the driver together and integrate the keyboard driver as a background task into polyFORTH. Line 1 in block 308 defines a background task. With **+kbd** the task is activated and with **-kbd** it is stopped.

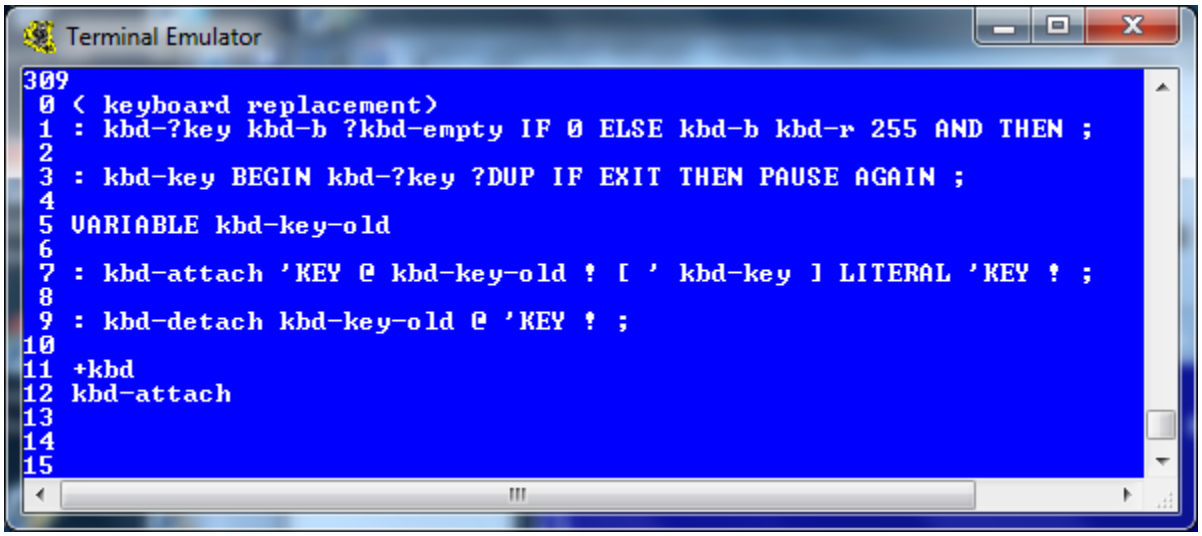


Listing 24 polyFORTH background task

The word **kbd-test** is a small test program. It loops and emits every key entered until the space bar is hit. It depends on the background task for running. Note that block 308 will not load for arrayFORTH versions prior to h.

More details about multitasking can be found in [4] and [5].

Block 309 is not loaded though block 300, because it replaces the terminal **'KEY** vector and switches any keyboard input to the PS/2 interface. Block access via the serial terminal is not possible while using the PS/2 keyboard. **kbd-detach** will restore the old vector, thus returning to the serial terminal for input.



Listing 25 Keyboard takeover

This driver is only an example of how to access a keyboard. If a software needs also the break codes from the keyboard, then the driver must be modified. If you have a keyboard with more keys than a standard us-layout, then you can very easily detect the scan codes (**tstr**) and modify the key map for these new keys.

Function keys can be attached easily to words (e.g. **' xxx 0 kbd-fun !** will attach the word xxx to the F1 key).

This driver only generates ASCII values used by the word **KEY**. It requires 4 nodes on the GA144 and about 2k cells (including 512 cells for the key map) in the polyFORTH dictionary. The buffers are managed by a polyFORTH background task.

5. Conclusion

First we demonstrated how to use the multi-computer paradigm of the GA144 chip to access a PS/2 device. Then we integrated the higher level aspects for handling a keyboard into the polyFORTH environment. The combination of nodes allocated for a specific task and the flexibility of the polyFORTH virtual machine gives the developer a high degree of freedom to design hardware and software.

The author of this document is a C++ programmer and began using the arrayFORTH environment and the EVB001 evaluation board about 6 months prior to this app note as a hobbyist. With the help of the documents and staff from GreenArrays it was possible to write the driver software in a few weeks. Although arrayFORTH and polyFORTH are very different than environments most programmers are used to, it is worthwhile to begin working with these GreenArrays tools.

This application note was finished during a vacation visit to GreenArrays in Incline Village, NV. The author wants to thank for the help and patience he received from the GreenArray staff.

6. References

[1] http://www.computer-engineering.org

[2] http://www.computer-engineering.org/ps2keyboard/

[3] http://www.computer-engineering.org/ps2keyboard/scancodes3.html

[4] “polyFORTH Reference manual”, GreenArrays Data Books: DB005

[5] “G144A12 polyFORTH supplement to DB005”, GreenArrays Data Books: DB06

[6] “F18A Technology Reference”, GreenArrays Data Books: DB001

GreenArrays®

Application Note AN009 *Revised 9/12/12*

IMPORTANT NOTICE

GreenArrays Incorporated (GAI) reserves the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to GAI's terms and conditions of sale supplied at the time of order acknowledgment.

GAI disclaims any express or implied warranty relating to the sale and/or use of GAI products, including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright, or other intellectual property right.

GAI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using GAI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

GAI does not warrant or represent that any license, either express or implied, is granted under any GAI patent right, copyright, mask work right, or other GAI intellectual property right relating to any combination, machine, or process in which GAI products or services are used. Information published by GAI regarding third-party products or services does not constitute a license from GAI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from GAI under the patents or other intellectual property of GAI.

Reproduction of GAI information in GAI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. GAI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of GAI products or services with statements different from or beyond the parameters stated by GAI for that product or service voids all express and any implied warranties for the associated GAI product or service and is an unfair and deceptive business practice. GAI is not responsible or liable for any such statements.

GAI products are not authorized for use in safety-critical applications (such as life support) where a failure of the GAI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of GAI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by GAI. Further, Buyers must fully indemnify GAI and its representatives against any damages arising out of the use of GAI products in such safety-critical applications.

GAI products are neither designed nor intended for use in military/aerospace applications or environments unless the GAI products are specifically designated by GAI as military-grade or "enhanced plastic." Only products designated by GAI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of GAI products which GAI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

GAI products are neither designed nor intended for use in automotive applications or environments unless the specific GAI products are designated by GAI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, GAI will not be responsible for any failure to meet such requirements.

For current information on GreenArrays products and application solutions, see www.GreenArrayChips.com

Mailing Address: GreenArrays, Inc., 774 Mays Blvd #10 PMB 320, Incline Village, Nevada 89451
Printed in the United States of America
Phone (775) 298-4748 fax (775) 548-8547 email Sales@GreenArrayChips.com