I²C EEprom with Driver for Extension Card for the K1-Bus

Version 1.12
Date 2013-04-13

This Document describes the layout of driver EEproms on K1-Bus peripheral cards and the bytecode used for universal drivers.

The K1-Bus is a simple 16-bit peripherals bus for home-built computers.
For a description of the K1-Bus hardware and circuit examples, see http://k1.spdns.de/Develop/Hardware/K1-Bus/.

The I²C EEproms on K1-Bus extension cards contain universal driver software in bytecode for a simple virtual machine. This universal driver may be used by the host system to install an extension card if it does not provide own drivers for the card in it's own BIOS.

Content

EEprom Layout

The EEprom is organized in *chunks of data*. Chunks start immediately at EEprom address 0 and follow each other without gap.

Data Types Used in Chunk Definitions:

int8 1 byte unsigned integer number in range 0 … 255
int12 1 or 2 bytes unsigned integer number in range 0 … $0FFF
1 byte = 0 … $EF or
2 bytes = n + $F000; high byte first.
TYPE[] Variable-length array of TYPE. Preceded with an **int12** length prefix.
str Text string: **int8[]** length-prefixed array of characters,
Ascii or *Latin-1*. (7- or 8-bit subset of Unicode.)

General Layout of Chunks:

A chunk starts with a 1-byte *chunk ID* followed by *length-prefixed data*:

int8	**BTYPE**	chunk ID
int8[]	**DATA**	length-prefixed data

Predefined Chunk IDs:

Chunks "0" – "5" and "A" – "D" must appear in this order.
* = mandatory block
** = may occur multiple times

BTYPE	"0"	*	EEprom identification
BTYPE	"1"	*	Device information: bit fields
BTYPE	"2"	*	Short device name for installation into a device directory
BTYPE	"3"	*	Short card name for display during boot process
BTYPE	"4"	*	Short card and driver version
BTYPE	"5"	*	Short copyright message
BTYPE	"6"		Longish license and copyright text or url
BTYPE	"7"		Longish information and help text or url
BTYPE	"A"	*	TYPEDEFS – Data type definitions
BTYPE	"B"	*	GLOBALS – Global variables
BTYPE	"C"	**	PROCDEF – Procedures and interrupt handler
BTYPE	"D"	*	End of data
BTYPE	"E"		Erased block
BTYPE	"F"		COMMAND – Loadable shell command

Reserved Chunk IDs:

BTYPE $00 – $5F (e.g. numbers and uppercase letters) are reserved.
BYTPE $60 – $7F (e.g. lowercase letters) may be used by the driver.

Chunk Descriptions

BTYPE "0" * EEprom Identification

Required. The first chunk in the EEprom identifies the EEprom.

```
BTYPE    '0'
DATA     "K1D1xxxx"   // length-prefixed string
```

Substring "K1" says it contains chunked driver data, substring "D1" says it is data format version 1. Then 4 arbitrary characters follow. These can be used by a system to quickly identify the board and determine whether the system has it's own drivers for this board.

BTYPE "1" * DEVINFO – Device information

Required. The second chunk provides technical information about the device.

```
BTYPE    '1'
DATA     int8[]       // bit fields and bytes
   DATA[0] DEVCLASS_1 = Device Class:
   bit.0   INPUTDEV   1 = Input Device
   bit.1   OUTPUTDEV  1 = Output Device
   bit.2   SERIALDEV  1 = Serial i/o
   bit.3   BLOCKDEV   1 = Addressable Block i/o
   bit.4   TERMINAL   1 = Potential Controlling Terminal
   bit.5   WIDEDEV    1 = uses int16[] buffers and 16-bit i/o
   bit.6   VIDEODEV   1 = Video Device
   bit.7   AUDIODEV   1 = Audio Device
   DATA[1] DEVCLASS_2 = Device Class:
   bit.0   KEYBOARD   1 = Keyboard
   bit.1   POINTER    1 = Pointer Device (Mouse, Joystick)
   bit.2   TIMERDEV   1 = Timer Device (System Timer)
   bit++              Reserved. set to 0
   DATA[2] DEVREQ = Device Requirements:
   bit.0   A4A5REQ    1 = Requires bus address lines A4 and A5
   bit.1   FASTBOUT   1 = Supports burst-mode block output
                      (no data hold time after strobe)
   bit++              Reserved. set to 0
   DATA[3] IOSPEED = minIOCycleTime[µsec]*128;
                      e.g. 128 = 1µs
   DATA[4] INTPRIO = interrupt priority hint:
                      -log2(maxIntLatency[sec]);
                      e.g. 0=1s, 10=1ms, 20=1µs
   DATA[5] SUBDEVICES = number of subdevices or channels
```

BTYPE "2" * Short device name

```
BTYPE    '2'
DATA     str      // e.g. "hd"
```

Required. This chunk provides a name for installation of the device into a device directory. Recommended names are "hd" for block devices and "sio" for serial devices.

BTYPE "3" * Short card name

```
BTYPE    '3'
DATA     str      // e.g. "FooCard"
```

Required. This name may be displayed during boot process.

BTYPE "4" * Short card and driver version

```
BTYPE    '4'
DATA     str      // e.g. "1.0a"
```

Required. The version number may be displayed during boot process.

BTYPE "5" * Short copyright message

```
BTYPE    '5'
DATA     str      // e.g. "(c) 2012-2013 kio@little-bat.de"
```

BTYPE "6" Longish license and copyright text or url

```
BTYPE    '6'
DATA     str[]    // Array of strings
```

This chunk contains license and copyright informations.
DATA is a 2-dimensional array of characters: **str[]** is a length-prefixed array of strings, which are length-prefixed arrays of characters (bytes).
The **str[]** array contains one paragraph per string. No line breaks required.

BTYPE "7" Longish information and help text or url

```
BTYPE    '7'
DATA     str[]    // Array of strings
```

This chunk contains usage and help informations.
The **str[]** array contains one paragraph per string. No line breaks required.

BTYPE "A" * TYPEDEFS – Data Type Definitions

Required. This chunk defines the custom struct data types used by the driver bytecode.

```
BTYPE    'A'
DATA     int8[][]   // Array of int8[] data member lists
```

Data types are referenced by small integer numbers in the bytecode and in the *TYPEDEFS*, *GLOBALS* and *PROCDEF* chunks.

This chunk contains a list of data member lists for all custom struct types. The first list defines the data members for struct type *ID = 7*, following lists define following type IDs.

Struct types are referenced in Bytecode by their position in this list.
Struct members are referenced by their position in their respective member list.

Predefined bytecode data types:

(don't mix these up with data types used for chunk definition)

00	void	
01	int8	unsigned integer: ≥8 bit
02	int16	unsigned integer: ≥16 bit
03	pointer	pointer to array or struct
04	int8[]	array of int8
05	int16[]	array of int16
06	pointer[]	array of arrays or structs (not used in Bytecode)
≥ 7	custom struct types as defined in this *TYPEDEFS* block	

void is used for procedures which do not return a value.
int8 (bytes) are at least 8 bit wide but may be longer. *int16* (words) are at least 16 bit wide but may be longer. Systems may actually use the same size for **int8** and **int16**. Values on the data stack are always **int16**.

Bytecode only supports *UNSIGNED* integer numbers.

Arrays like *text strings* and *structs* are allocated dynamically and are referenced by *pointers*.

For more information on data types see the section about Bytecode below.

BTYPE "B" * GLOBALS – Global Variables

Required. This chunk defines the global variables used by the driver.

```
BTYPE    'B'
DATA     int8[]       // List of data types
```

Global variables are referenced in Bytecode by their position in this list.

Data types are 0 to 6 for the predefined basic types and data types ≥ 7 are assigned to custom struct types as defined in the *TYPEDEFS* chunk.

Global data is initially cleared with 0. Arrays and structs must be allocated explicitly in function **init()**.

*Global data must start with **n** struct variables, with **n** = number of channels or subdevices,* which should contain all data for the channel. A reference to one of these structs is the first argument in most *public* procedures of the drivers.

BTYPE "C" ** PROCDEF – Procedure or Interrupt Handler

```
BTYPE  'C'
DATA   int12    block length
       int8     Procedure reference number (ID)
       int8     return type: void or int16
       int8[]   Type list of procedure arguments
       int8[]   Type list of more local variables
       int8[]   Bytecode
```

Required, multiple. Each *PROCDEF* chunk defines one procedure. It defines a function reference number, a return value type, an arguments type list, a local variables type list and the bytecode for this procedure.

Reference numbers must be unique. *Procedures are referenced in Bytecode by their reference number (ID).* A procedure can only call procedures which have already been defined. For each device type there are some predefined *public* functions with IDs in range 0 to 31. *Internal* functions may use IDs in range 32 to 127.

The return type may be either **void** or **int16**.

Arguments and local variables may be of any type except **void**.

The arguments type list and the local variables type list are concatenated by the bytecode loader to form a combined local variables list. *Local variables are referenced in Bytecode by their position in this combined list.*

BTYPE "D" * End of Data

```
BTYPE    'D'
DATA     int[0]
```

Required. This chunk marks the end of valid data in the EEprom. Any data beyond this chunk is void. The *DATA* consists only of a length prefix which is 0.

BTYPE "E" Erased Data

```
BTYPE    'E'
DATA     int8[]
```

This chunk can be used to erase a chunk without moving all subsequent chunks. Also, if you overwrite a chunk with a shorter chunk, an "erased data" block can be used to fill the remainder. If the new chunk is exactly 1 byte shorter try to use a 2-byte length descriptor for the new data.

BTYPE "F" COMMAND – Shell Command etc.

```
BTYPE    'F'
DATA     int12    block length
         str      Procedure name
         int8     return type: void or int16
         int8[]   Type list of procedure arguments
         int8[]   Type list of local variables
         int8[]   Bytecode
```

COMMANDs are not loaded during boot time but may be loaded and executed at runtime to provide useful functions. The return value should in general be an error code with 0 for ok. See *PROCDEF* for description of the contents.

COMMANDs should follow after the last *PROCDEF* chunk.

COMMANDS are still under development.

Predefined Functions

Function definition syntax: *<procID>*: *<return_type>* *<name>* (*<arguments>*)

0: void init ( )

Required. All devices must have an init function. **init()** is the last procedure loaded by the boot loader.

After loading **init()**, global data is cleared with zero and then **init()** is called to initialize the hardware and to allocate and initialize global arrays and struct variables. Eventually some other initialization functions are called after **init()**, e.g. **systemtimer()**.

After that, all *disposable* functions are purged. All procedures, which are only called during initialization are *disposable*. These are the *public* procedures **init()** and **systemtimer()** and all *internal* procedures which are only called by these. i.e. *internal* functions (32 ≤ ID ≤ 127) are *disposable*, if they were defined after the last non-*disposable* *public* function.

init() is called with the device selected and interrupts turned off.

8: void irpt ( )

Most devices generate interrupts. **irpt()** is called by the system to handle these interrupts.

If a device signals an interrupt, then the system disables interrupts, selects the device and calls the device's interrupt handler function **irpt()**. On return, it deselects the device, enables interrupts and returns to the interrupted main program.

When the driver code calls **irpt()** directly, e.g. to restart output of a serial device, then the bytecode loader must add *di* before and *ei* after the call, if it does not add these opcodes to the **irpt()** procedure itself.

Note: interrupts become enabled after calling **irpt()**.

All i/o on the K1-Bus goes to the *currently selected device*. The i/o address, which is part of the *in* and *out* opcodes, is only used to select registers in the device.

irpt() may be called erroneously.
irpt() is called with the device selected and interrupts turned off.
irpt() must effectively switch off the hardware interrupt.
irpt() must not allocate or dispose memory, therefore it cannot allocate local arrays or structs.

Device Types

So far, the following device types have been defined:

- *System Timer*, *defined in version 1.0*
- *Serial Device* and *defined in version 1.0*
- *Block Device*. *defined in version 1.0*

Keyboard, pointer and audio will be based on serial, video on block devices.

System Timer Device

A timer device is identified by bit *TIMERDEV* in *DEVINFO*. A timer device can provide a regular system timer interrupt.

A Timer Device must provide:

1: int systemtimer (int period_µs)

If the system has no own system timer, it will call **systemtimer()** after **init()** but before all disposable functions are destroyed.
The argument is the desired timer period in *microseconds*.
Legal values are in range 1000 (1000Hz) to 20000 (50Hz).
systemtimer() sets up the timer interrupt and returns the actually used timer period in *microseconds*.

systemtimer() is called with the device selected and interrupts turned off.

The **irpt()** function must execute the *timer* opcode once per timer interrupt if the timer was enabled by a call to **systemtimer()**, else the device should not generate the timer interrupt at all.

Serial Device

A serial device is identified by bit *SERIALDEV* in *DEVINFO*. A serial device provides buffered serial input and/or output of data. The number of channels is defined in *DEVINFO* in byte *SUBDEVICES*. Serial devices may be 8 or 16 bit wide. Bit *WIDEDEV* in *DEVINFO* identifies a 16-bit device. Most serial devices are 8 bit wide.

Note: **channel** is one of the first *n* structs *(n = number of subdevices)* in the device's global variables. **data** is **int8** or **int16** depending on device size.

A Serial Device must provide:

9: void setctl (channel, int function_code, int value)

setctl functions:

00	reset channel
01	set serial speed to *value* * 100
02	set HW handshake on/off
03	set SW handshake on/off
04	purge (clear) input buffer
05	purge (clear) output buffer

10: int getctl (channel, int function_code)

getctl functions:

01	get serial speed / 100
02	get HW handshake state
03	get SW handshake state
04	get available (non-blocking) bytes in input buffer
05	get available (non-blocking) space in output buffer

A Serial Input Device must provide:

11: int getc (channel)

Read one byte from the serial *channel*.
Blocking!

12: int gets (channel, data[], int a, int e)

Read *data[]* from index *a* to *e-1* from the serial *channel*.
a and *e* must be in range *0* ≤ *a* ≤ *e* ≤ *data.count*. *e* may be == *a*.
May read less. Returns actual number of bytes read.
Non-blocking.

A Serial Output Device must provide:

13: void putc (channel, int char)

Write one byte to the serial *channel*.
Blocking!

14: int puts (channel, data[], int a, int e)

Write *data[]* from index *a* to *e-1* to the serial *channel*.
a and *e* must be in range *0* ≤ *a* ≤ *e* ≤ *data.count*. *e may be* == *a*.
May write less. Returns actual number of bytes written.
Non-blocking.

A Controlling Terminal must provide:

- Serial Input
- Serial Output

A potential controlling terminal is identified by bit *TERMINAL* in *DEVINFO*.
Eventually one data direction may be missing if two disjunct serial lines are used for keyboard and monitor.

Block Device

A block device is identified by bit *BLOCKDEV* in *DEVINFO*. A block device provides i/o of addressable blocks of data of a fixed size. Block devices may be 8-bit or 16-bit devices, accordingly with **int8[]** or **int16[]** buffers. A 16-bit block device is identified by bit *WIDEDEV* in *DEVINFO*. The number of subdevices is defined in byte *SUBDEVICES* in *DEVINFO*. A block device may contain up to 2^32-1 blocks. The block number is split into a high and a low **int16** value.

Note: **subdev** is one of the first *n* structs *(n = number of subdevices)* in the device's global variables. **data** is **int8** or **int16** depending on the device size.

A Block Device must provide:

9: void setctl (subdev, int function_code, int value)

setctl functions:

00	reset subdevice

10: int getctl (subdev, int function_code)

getctl functions:

07	get log2(blocksize)	note: blocksize must be 2^N
08	get total blocks (low)	
09	get total blocks (high)	

15: int readblocks (subdev, int start_hi, int start_lo, data[], int a, int e)

16: int writeblocks (subdev, int start_hi, int start_lo, data[], int a, int e)

Read or write *data[]* from index *a* to *e-1* from or to *subdev*ice.
a and *e* must be in range *0* ≤ *a* ≤ *e* ≤ *data.count*.
e-a must be a multiple of *blocksize* and may be 0.
start_hi + *start_lo* define the starting block number.
Blocking!

Note: Depending on the byte order on the data stack the target system may actually use one **int32** for the *start*ing block number when calling **readblocks()** or **writeblocks()**.

Returns error code:

0	ok
1	parameter error / function not supported
2	io error
3	device not responding / device not present

Summary of ioctl function numbers:

0	set	ioctl_reset
1	set/get	ioctl_sio_speed
2	set/get	ioctl_sio_hw_hsk
3	set/get	ioctl_sio_sw_hsk
4	set	ioctl_sio_purge_input
5	set	ioctl_sio_purge_output
4	get	ioctl_sio_input_avail
5	get	ioctl_sio_output_free
6		ioctl_sio_reserved1
7	get	ioctl_blk_log2blocksize
8	get	ioctl_blk_totalblocks_lo
9	get	ioctl_blk_totalblocks_hi
10		ioctl_blk_reserved2

Proc IDs of predefined public procedures:

Proc IDs 0 to 31 are reserved for predefined public procedures.
Proc IDs 0 to 7 are disposable.

0	void	init ()	*disposable*
1	int	systemtimer (int µs)	*disposable*
8	void	irpt ()	
9	void	setctl (channel, int function_code, int value)	
10	int	getctl (channel, int function_code)	
11	int	getc (channel)	
12	int	gets (channel, data[], int a, int e)	
13	void	putc (channel, int char)	
14	int	puts (channel, data[], int a, int e)	
15	int	readblocks (subdev, int start_hi, int start_lo, data[], int a, int e)	
16	int	writeblocks (subdev, int start_hi, int start_lo, data[], int a, int e)	

Bytecode

The I²C EEproms on K1-Bus extension cards contain universal driver software in bytecode for a simple virtual machine.

Bytecode instruction are one byte each. Some instructions have inline arguments. They are designed to allow simple 1:1 translation into short code snippets. This can either be real machine code for the target system's CPU or code for a forth-style interpreter.

Supported data types in memory are **int8** (unsigned bytes ≥ 8 bit), **int16** (unsigned words ≥ 16 bit), **pointers** to dynamically allocated **int8** or **int16** **arrays**, and **pointers** to dynamically allocated custom **struct** types. The bytecode supports memory models with movable memory.

A separate data stack is used for local variables and intermediate values. It is probably possible to compile Bytecode into register-based machine code, but this is more tricky. Values on the data stack are **int16** or **pointers**, which may be of a different size. **int8** data must be padded with 0 when put on the stack.

For simplicity, returning **pointers** from procedures is not supported.

Variables come in four styles: **global**, **local**, **array item** and **struct member**. Variable accessors are followed by a variable reference which specifies which global or local variable to access, or by a struct type reference and a struct member reference in case of structs.

For an example of translating a high-level source into bytecode see:

compiler: http://k1.spdns.de/Develop/Projects/vicci
driver: http://k1.spdns.de/Develop/Hardware/K1-Computer/IO-Boards/SIO

Data Types

Data types as used in *TYPEDEFS*, *GLOBALS* and *PROCDEF* chunks and for data type reference ***T*** in some opcodes:

00	void	
01	int8	unsigned integer: ≥ 8 bit
02	int16	unsigned integer: ≥ 16 bit
03	pointer	
04	int8[]	array of int8
05	int16[]	array of int16
06	pointer[]	array of arrays or structs (not used in Bytecode)
≥ 7	custom struct types as defined in the *TYPEDEFS* chunk	

void is used to declare the return type of procedures which return no value.

int8 'bytes' are at least 8 bit wide but may be longer. *int16* 'words' are at least 16 bit wide but may be longer. Systems may actually use the same size for **int8** and **int16**. See 'Notes on using same size for int8 and int16'. Bytecode only supports *UNSIGNED* integer numbers.

*Values on the data stack are always **int16**.* A procedure argument may be defined **int8**. Then it is read and written as **int8** inside the function, but it is passed as **int16** on the data stack.

Arrays, e.g. *text strings,* and *structs* must be allocated dynamically and are referenced by *pointers*. Dynamically allocated data may be moved around by the system. The size of arrays is not part of the data type; it is part ot the data. For simplicity, returning **pointers** from procedures is not supported.

int8[] arrays can be sent to and received from a device via 8-bit i/o opcodes.

int16[] arrays can be sent to and received from a device via 16-bit i/o opcodes.

Data Access

Data setters and getters for *local* and *global* variables are followed by a byte which contains the variable reference. The variable reference consists of the variable's index in the *globals* or *locals* list. Indexes always start at ***0***.

The *local* variables list is combined from the procedure's arguments and local variables declarations as defined in the *PROCDEF* chunk.

The *global* variables are defined in the *GLOBALS* chunk.

The target system must construct a list of *global* variables' types from the *GLOBALS* chunk, and a list of *local* variables' types for each procedure. Then for every data accessor opcode it must compile the appropriate address or offset to access the variable.

Data setters and getters for *struct* members are followed by one byte which defines the struct type and one byte which defines the struct member by it's index in the struct's data member list as defined in the *TYPEDEFS* chunk.

Data setters and getters for *array* items are not followed by inline data. Instead there are 2 distinct opcodes for **int8** and **int16 arrays**. The array variable itself and the index are passed as arguments on the data stack.

There is no opcode intended to access random memory with a pointer.

K1-Bus Data I/O:

Devices on the K1-Bus are selected with the *di* opcode, which disables interrupts and selects the device.

i/o instructions do not need to know the address of their device. All i/o on the K1-Bus goes to the *currently selected device*. The i/o address, which is part of the *in* and *out* opcodes, is only used to select registers in the device.

The system must provide 8-bit and 16-bit variants of each i/o opcode, except if it does not support 16-bit i/o at all.

The I²C EEprom contains a flag in chunk *DEVINFO* for **burst-mode** block output. Normally the K1-Bus timing chart defines a data hold time after the data strobe goes inactive. Eventually the target system can increase i/o speed, if this requirement is relaxed for block outputs. (inputs aren't affected anyway.) If bit

FASTBOUT is set in the *DEVINFO* chunk, then the target system can use block i/o instructions with very little or no data hold time after strobe, only asserting that the data is stable at least until when the i/o strobe goes inactive.

Abbreviations for arguments used in the bytecode descriptions:

Most arguments and return values are passed on the data stack. Sometimes fixed arguments are passed inline after the opcode. This is indicated by green text color:

Value on stack:

- p = **pointer**: array, struct, address of variable
- n = **int16** value on stack
- i = **int16** value on stack used as index
- a = **int16** value on stack used as address
- n1 = **int16** value on stack with a reference for comment

Pointers on stack:

- &b = pointer to **int8** 'byte' variable
- &w = pointer to **int16** 'word' variable
- &p = pointer to **pointer** variable: **array** or **struct**
- &x = pointer to **b**, **w** or **p** variable
- b[] = pointer to **int8 array**
- w[] = pointer to **int16 array**

Inline argument:

- **T** = inline **int8**: type ID as defined in the *TYPEDEFS* chunk
- **L** = inline **int8**: label
- **b** = inline **int8**: value
- **w** = inline **int16** value, high byte first
- **b[]** = inline **int8 array** with 1 byte length prefix:
  used for procedure names and **int8 array** initializer
- **w[]** = inline **int16 array** with 1 byte length prefix, high bytes first:
  used for **int16 array** initializer
- **L[]** = inline **int8 array** with 1 byte length prefix: labels

Examples:

ati.b (b[] i -- &b) // *get pointer to item in **int8 array** at index i*

Input: pointer to **int8 array**
int16 index
Output: **pointer** to **int8** data

lvar (**b** -- &x) // *get pointer to local variable*

Input: inline **int8** index in *local* variables table
Output: **pointer** to **int8**, **int16** or **pointer** variable

Opcodes Sorted by Code

Push immediate value:

$00	ival.b	(**b** -- n)	push **int8** value, 0 … 255
$01	ival.w	(**w** -- n)	push **int16** value, high byte first
$02	ival.b[]	(**b[]** -- b[])	push length-prefixed **int8 array**
$03	ival.w[]	(**w[]** -- w[])	push length-prefixed **int16 array**

Arithmetics & logics on stack:

$04	swap	(n -- n)	swap high byte and low byte of word
$05	get_lo	(n -- n)	get low byte of word
$06	get_hi	(n -- n)	get high byte of word
$07	cpl	(n -- n)	~n
$08	not	(n -- n)	n ? 0 : 1
$09	msbit	(n -- n)	log2(n)
$0A	add	(n n -- n)	
$0B	sub	(n1 n2 -- n)	n1 – n2
$0C	mul	(n n -- n)	
$0D	div	(n1 n2 -- n)	n1 / n2
$0E	rem	(n1 n2 -- n)	n1 % n2
$0F	and	(n n -- n)	
$10	or	(n n -- n)	
$11	xor	(n n -- n)	
$12	sl	(n1 n2 -- n)	n1 << n2
$13	sr	(n1 n2 -- n)	n1 >> n2
$14	eq	(n n -- n)	
$15	ne	(n n -- n)	
$16	lt	(n1 n2 -- n)	n1 < n2
$17	le	(n1 n2 -- n)	n1 ≤ n2
$18	gt	(n1 n2 -- n)	n1 > n2
$19	ge	(n1 n2 -- n)	n1 ≥ n2
$1A	min	(n n -- n)	

Access variables:

$1B	lvar	(**b** -- **&x**)	get pointer to *local* variable
$1C	gvar	(**b** -- **&x**)	get pointer to *global* variable
$1D	ivar	(p **T** **b** -- **&x**)	get pointer to member **b** in **struct** *p* of type **T**
$1E	ati.b	(b[] i -- **&b**)	get pointer to item in **int8 array** *b[]* at index *i*
$1F	atiget.b	(b[] i -- n)	get item from **int8 array** *b[]* at index *i*
$20	atiset.b	(n b[] i --)	set item in **int8 array** *b[]* at index *i*
$21	lget.b	(**b** -- n)	get *local* **int8** variable
$22	lset.b	(n **b** --)	set *local* **int8** variable
$23	gget.b	(**b** -- n)	get *global* **int8** variable
$24	gset.b	(n **b** --)	set *global* **int8** variable
$25	iget.b	(p **T** **b** -- n)	get **int8** *member* **b** in **struct** *p* of type **T**
$26	iset.b	(n p **T** **b** --)	set **int8** *member* **b** in **struct** *p* of type **T**
$27	addgl.b	(n **&b** --)	add value to **int8** variable etc.
$28	subgl.b	(n **&b** --)	
$29	andgl.b	(n **&b** --)	
$2A	orgl.b	(n **&b** --)	
$2B	peekpp.b	(**&b** -- n)	get value from **int8** variable with post-incr.
$2C	mmpeek.b	(**&b** -- n)	get value from **int8** variable with pre-decr.
$2D	ati.w	(w[] i -- **&w**)	get ptr. to item in **int16 array** *w[]* at index *i*
$2E	atiget.w	(w[] i -- n)	get item from **int16 array** *w[]* at index *i*
$2F	atiset.w	(n w[] i --)	set item in **int16 array** *w[]* at index *i*
$30	lget.w	(**b** -- n)	get *local* **int16** variable
$31	lset.w	(n **b** --)	set *local* **int16** variable
$32	gget.w	(**b** -- n)	get *global* **int16** variable
$33	gset.w	(n **b** --)	set *global* **int16** variable
$34	iget.w	(p **T** **b** -- n)	get **int16** *member* **b** in **struct** *p* of type **T**
$35	iset.w	(n p **T** **b** --)	set **int16** *member* **b** in **struct** *p* of type **T**
$36	addgl.w	(n **&w** --)	add value to **int16** variable etc.
$37	subgl.w	(n **&w** --)	
$38	andgl.w	(n **&w** --)	
$39	orgl.w	(n **&w** --)	
$3A	peekpp.w	(**&w** -- n)	get value from **int16** variable with post-incr.
$3B	mmpeek.w	(**&w** -- n)	get value from **int16** variable with pre-decr.
$3C	lget.p	(**b** -- p)	get **pointer** from *local* **pointer** variable
$3D	gget.p	(**b** -- p)	get **pointer** from *global* **pointer** variable
$3E	iget.p	(p **T** **b** -- p)	get **pointer** member **b** in **struct** *p* of type **T**
$3F	set.p	(p **&p** --)	store **pointer** in **pointer** variable
$40	not.p	(p -- n)	test pointer for null

Arrays & memory:

$41	alloc.b	(n -- b[])	allocate **int8 array**, cleared with 0
$42	alloc.w	(n -- w[])	allocate **int16 array**, cleared with 0
$43	alloc.S	(**T** -- p)	allocate **struct** of type **T**, cleared with 0
$44	count.b	(b[] -- n)	get item count in **int8 array**
$45	count.w	(w[] -- n)	get item count in **int16 array**
$46	copy.b	(b[] i1 b[] i2 n)	copy **array** *b[i1 to i1+n-1]* to *b[i2 to i2+n-1]*
$47	copy.w	(w[] i1 w[] i2 n)	copy **array** *w[i1 to i1+n-1]* to *w[i2 to i2+n-1]*
$48	dispose	(p --)	dispose **array** or **struct**

Input and output to the K1-Bus:

$49	in.b	(a -- n)	input **int8** from i/o address *a*
$4A	out.b	(n a --)	output **int8** to i/o address *a*
$4B	bin.b	(b[] i n a --)	input **int8 array** *b[i to i+n-1]* from i/o addr. *a*
$4C	bout.b	(b[] i n a --)	output **int8 array** *b[i to i+n-1]* to i/o address *a*
$4D	in.w	(a -- n)	input **int16** from i/o address *a*
$4E	out.w	(n a --)	output **int16** to i/o address *a*
$4F	bin.w	(w[] i n a --)	input **int16 array** *w[i to i+n-1]* from i/o addr. *a*
$50	bout.w	(w[] i n a --)	output **int16 array** *w[i to i+n-1]* to i/o addr. *a*
$51	readi2c	(a b[] i n --)	read *b[i to i+n-1]* from I²C EEprom addr. *a++*
$52	writei2c	(a b[] i n --)	write *b[i to i+n-1]* to I²C EEprom address *a++*

Interrupts:

$53	di	(--)	disable interrupts & select device
$54	ei	(--)	deselect device & enable interrupts
$55	timer	(--)	call the system timer handler
$56	wait	(--)	halt CPU & wait for interrupt
$57	systemtime	(-- n)	get low **int16** of system time; approx. in ms

Code Flow:

$58	jp	(**L** --)	jump to label **L**
$59	jp0	(**L** n --)	jump to label **L** if *n* is FALSE
$5A	jp1	(**L** n --)	jump to label **L** if *n* is TRUE
$5B	and0	(**L** n --)	keep *n* and jump to **L** if *n* is FALSE, else drop *n* & don't jump
$5C	or1	(**L** n --)	keep *n* and jump to **L** if *n* is TRUE, else drop *n* & don't jump
$5D	switch	(**L[]** n --)	jump to label **L[*n*]**
$5E	call	(**b** --)	call procedure with proc ID **b**
$5F	ret	(--)	return from procedure
$60	label	(**L** --)	pseudo opcode: define label **L** here.

Discussion of all Bytecode Opcodes

Push immediate value:

ival.b (b -- n) *push* ***int8*** *value*
ival.w (w -- n) *push* ***int16*** *value*

Push immediate value on the data stack. The value follows inline after the opcode.

The **int16** value is stored high byte first. If the target system prefers it the other way then the bytecode loader may just swap the two bytes. The **int8** value is in range 0 to 255, **int16** in range 0 to 65535.

ival.b[] (b[] -- b[]) *push length-prefixed* ***int8 array****; e.g. a text string*
ival.w[] (w[] -- w[]) *push length-prefixed* ***int16 array***

Push an array of bytes (**int8**) or words (**int16**) on the data stack. The byte or word array follows inline after the opcode and is preceded by a 1-byte length prefix. Words in the **int16 array** are stored high bytes first and may be swapped by the bytecode loader if required.

When this opcode is executed, the system *should* dynamically allocate the required amount of memory and initialize it with the inline data. The array must be disposable with the *dispose* opcode, but it is considered *const* and should not be overwritten by the driver. These opcodes are preferably used for initialization of tables in a disposable procedure.

Arithmetics and logics on stack:

add (n1 n2 -- n) ***n1 + n2***
sub (n1 n2 -- n) ***n1 – n2***
mul (n1 n2 -- n) ***n1 * n2***
div (n1 n2 -- n) ***n1 / n2***
rem (n1 n2 -- n) ***n1 % n2***

The 5 basic arithmetic operations.

Note the order of arguments for *sub*, *div* and *rem*.

Systems with unusual **int16** size must mask the result of *add*, *sub* and *mul* and argument ***n1*** in *div* and *rem* with $FFFF.

```
n  =  (n1 + n2) & $FFFF          // add
n  =  (n1 & $FFFF) / n2          // div
```

Note: *mul*, *div* and *rem* may be longish operations on small systems and should be used only if neccessary. Multiplication and division can often be replaced by bit shifting with *sl* and *sr*, remainder by masking with *and*. Therefore it is recommended to make all buffers *2^N* in size.

and (n n -- n)
or (n n -- n)
xor (n n -- n)

The 3 basic bit operations.

sl (n1 n2 -- n) n1 << n2
sr (n1 n2 -- n) n1 >> n2

Shift bits left or right, padding with 0. Shift distance *n2* is masked with $0F. Systems with unusual **int16** size must mask the value before *sr* or after *sl* with $FFFF.

```
n = (n1 << (n2&$F)) & $FFFF        // sl
n = (n1 & $FFFF) >> (n2&$F)        // sr
```

eq (n1 n2 -- n) n1 == n2
ne (n1 n2 -- n) n1 != n2
lt (n1 n2 -- n) n1 < n2
le (n1 n2 -- n) n1 ≤ n2
gt (n1 n2 -- n) n1 > n2
ge (n1 n2 -- n) n1 ≥ n2

The 6 basic compare operators.
The result is 0 for FALSE and 1 for TRUE.

get_lo (n -- n) *get low byte of word*
get_hi (n -- n) *get high byte of word*

Get lower 8 bits resp. upper 8 bits from **int16** value.
The result is in range 0 to 255:

```
n = n & $FF          // get_lo
n = (n>>8) & $FF     // get_hi
```

cpl (n -- n)

Bitwise complement of value.
Systems with unusual **int16** size must mask the result with $FFFF.

```
n = ~n & $FFFF
```

not (n -- n) *n ? 0 : 1*

Boolean negation. The result is 0 for FALSE and 1 for TRUE.

swap (n -- n)

Swap high and low byte of word:

```
n = (n & $FF) << 8 + (n>>8) & $FF
```

min (n1 n2 -- n) *n1<n2 ? n1 : n2*

Return the smaller value of *n1* and *n2*.

msbit (n -- n) *log2(n)*

Determine the most significant '1' bit in *n*.
The result is in range 0 to 15. The result for $0000 and for $0001 is 0.

Access variables:

lvar (b -- &x) *get pointer to local variable*

Get pointer to a *local* variable, which may be an **int8**, **int16** or a **pointer** for arrays or structs.

The inline byte is the index in the list of all *local* variables. This list is combined from the function arguments and variables defined inside the function. Indexes start at 0. See description of the *PROCDEF* chunk.

gvar (b -- &x) *get pointer to global variable*

Get pointer to a *global* variable, which may be an **int8**, **int16** or a **pointer** for arrays or structs.

The inline byte is the index in the list of all *global* variables as defined in the *GLOBALS* chunk. Indexes start at 0.

ivar (p T b -- &x) *get pointer to member b in struct p of type T*

Get pointer to a data *member* of a **struct**, which may be an **int8**, **int16** or a **pointer** for an array or struct.

The inline byte ***b*** is the index in the list of data members as defined in the *TYPEDEFS* chunk for the struct data type ***T.*** Member indexes start at 0.

Note: All opcodes which return a reference to a variable in allocted memory must be followed by the consumer opcode immediately.

ati.b (b[] i -- &b) *get pointer to item in byte array b[] at index i*
ati.w (w[] i -- &w) *get pointer to item in word array w[] at index i*

Get pointer to data in array at index *i*. Indexes start at 0. There separate opcodes for **int8** and **int16 arrays**.

Note: All opcodes which return a reference to a variable in allocted memory must be followed by the consumer opcode immediately.

atiget.b (b[] i -- n)
atiget.w (w[] i -- n)
atiset.b (n b[] i --)
atiset.w (n w[] i --)

Read or write data to array at index *i*. Indexes start at 0. There are getters and setters for **int8** and **int16**.

Note the 'reverse order' of arguments of the *atiset* opcodes.

lget.b ***(b -- n)***
lget.w ***(b -- n)***
lget.p ***(b -- p)***
lset.b ***(n b --)***
lset.w ***(n b --)***

Read or write data to *local* variable. There are getters and setters for **int8** and **int16**. For **pointers** only a getter is defined. For the setter *use* *lvar.p* plus *set.p*.

The inline byte is the index in the list of all *local* variables. This list is combined from the function arguments and local variables. Indexes start at 0. See description of the *PROCDEF* chunk.

Note the 'reverse order' of arguments of the *lset* opcodes.

gget.b ***(b -- n)***
gget.w ***(b -- n)***
gget.p ***(b -- p)***
gset.b ***(n b --)***
gset.w ***(n b --)***

Read or write data to *global* variable. There are getters and setters for **int8** and **int16**. For **pointers** only a getter is defined. For the setter *use* *gvar.p* plus *set.p*.

The inline byte is the index in the list of all *global* variables as defined in the *GLOBALS* chunk. Indexes start at 0.

Note the 'reverse order' of arguments of the *gset* opcodes.

iget.b ***(p T b -- n)***
iget.w ***(p T b -- n)***
iget.p ***(p T b -- p)***
iset.b ***(n p T b --)***
iset.w ***(n p T b --)***

Read or write data member in *struct* of type ***T***. There are getters and setters for **int8** and **int16** data members. For **pointer** data members only a getter is defined. For the setter *use* *ivar.p* plus *set.p*.

The inline byte ***b*** is the index in the list of data members as defined in the *TYPEDEFS* chunk for the struct data type ***T.*** Member indexes start at 0.

Note the 'reverse order' of arguments for the *iset* opcodes.

set.p (p &p --)

Store **pointer** in **pointer** variable. Pointers are **int16** on small systems and **int32** on most others. Setters for pointer variables must be combined from *lvar*, *gvar* or *ivar* and *set.p*.

Note the 'reverse order' of arguments.

not.p (p -- n) *test pointer for null*

Test pointer for **null**.

The result is 1 for TRUE (*p* is null) and 0 for FALSE.

Note: Bytecode contains no opcode to compare two pointers, only *not.p* to test a pointer for **null**.

peekpp.b (&b -- n) *get value from int8 variable with post-increment*
peekpp.w (&w -- n) *get value from int16 variable with post-increment*
mmpeek.b (&b -- n) *get value from int8 variable with pre-decrement*
mmpeek.w (&w -- n) *get value from int16 variable with pre-decrement*

Read and post-increment or pre-decrement and read **int8** or **int16** variable which is passed by reference.

addgl.w (n &w --) *add value to int16 variable etc.*
subgl.w (n &w --)
andgl.w (n &w --)
orgl.w (n &w --)
addgl.b (n &b --) *add value to int8 variable etc.*
subgl.b (n &b --)
andgl.b (n &b --)
orgl.b (n &b --)

Modify **int8** or **int16** variable which is passed by reference. Only a small subset of arithmetic operartions is supported for the 'modify' opcodes.

Note the 'reverse order' of arguments.

Arrays and memory:

alloc.b *(n -- b[]) allocate int8 array, cleared with 0*
alloc.w *(n -- w[]) allocate int16 array, cleared with 0*
alloc.S *(T -- p) allocate struct of type T, cleared with 0*

Allocate **int8** or **int16 array** with *n* items or a **struct** of type ***T***. Type ***T*** is passed inline in 1 byte and refers to the type ID as defined in the *TYPEDEFS* chunk. The newly allocated memory is cleared with 0.

dispose (p --) *dispose allocated data*

Dispose memory referenced by pointer *p*, either an **array** or a **struct**.
If a struct itself contains allocated data, this is not automagically disposed too. It must be actively disposed by the program before disposing the enclosing struct.

Pointer *p* may be *null*.

Note: Do not dispose data which you have not allocated yourself; e.g. arguments to a procedure. This will likely mix up the memory management of the host system. See 'Notes on dynamic memory'.

dispose is mostly used to destroy allocated local variables before the procedure *ret*urns.

count.b *(b[] -- n) get item count*
count.w *(w[] -- n) get item count*

Get number of items in **int8** or **int16 array**.

copy.b *(b[] i1 b[] i2 n) copy b[i1 to i1+n-1] to b[i2 to i2+n-1]*
copy.w *(w[] i1 w[] i2 n) copy w[i1 to i1+n-1] to w[i2 to i2+n-1]*

Copy range of bytes or words from one array to another. The first array and index *i1* are the source while the second array and *i2* are the destination. The number of bytes or words to copy is *n*.

Indexes and *n* must be in range *0* to *array.count*; *n* may be 0.

To support cyclic buffers, *i1+n* or *i2+n* may exceed beyond the end of the buffer and will wrap around to the start of the buffer during copy.

If source and destination buffer are the same, then *copy* must copy data with incrementing index, so that *copy* can be used if end of destination and start of source overlap. (copy to lower address.)

'copy' cannot be used if destination start and source end overlap.

Note: in a cyclic buffer source and destination can overlap at both ends! In general, *copy* will only be used to copy between different buffers, so this limitation to *copy* seems acceptable and simplifies implementation.

Input & output data to the K1-Bus:

Note: i/o is always done to the *currently selected device*. The address *a* is only used to select registers etc. in the device. To select the device, interrupts must be disabled with *di*.

K1-Bus i/o and EEprom access is only possible while interrupts are disabled!

A small system may implement an 8 bit wide K1-Bus only. Then the bytecode loader can check bit *WIDEDEV* in chunk *DEVINFO* of the driver EEprom to see whether this device requires a 16-bit bus and reject the device.

in.b ***(a -- n)*** *input* ***int8*** *from i/o address a*
in.w ***(a -- n)*** *input* ***int16*** *from i/o address a*
out.b ***(n a --)*** *output* ***int8*** *to i/o address a*
out.w ***(n a --)*** *output* ***int16*** *to i/o address a*

Input or output one **int8** byte or one **int16** word to or from the device.

in.b must clear the upper byte of *n* to 0.

bin.b ***(b[] i n a --)*** *input* ***int8 array*** *b[i to i+n-1] from i/o address a*
bout.b ***(b[] i n a --)*** *output* ***int8 array*** *b[i to i+n-1] to i/o address a*
bin.w ***(w[] i n a --)*** *input* ***int16 array*** *w[i to i+n-1] from i/o address a*
bout.w ***(w[] i n a --)*** *output* ***int16 array*** *w[i to i+n-1] to i/o address a*

Input or output block of **int8** or **int16** data to or from the device.

i and *n* must be in range $0 \le i \le i+n \le array.count$. *n* may be 0.

A device may support a faster 'burst' mode for output. This is declared in bit *FASTBOUT* in the *DEVINFO* chunk.

readi2c ***(a b[] i n -- n)*** *read* ***int8 array*** *b[i to i+n-1] from I²C EEprom*
writei2c ***(a b[] i n -- n)*** *write* ***int8 array*** *b[i to i+n-1] to I²C EEprom*

These opcodes allow to read and write all data in the currently selected driver EEprom starting at address *a*.

i and *n* must be in range $0 \le i \le i+n \le array.count$. *n* may be 0.

In general, these opcodes are not needed and should be used rarely, because i/o to the EEprom via the I²C bus is slow and blocking. They are provided so that utility functions can be written in bytecode.

Returns error code: same codes as for BlockDevice

0	ok
1	parameter error / function not supported
2	io error
3	device not responding / device not present

Interrupts:

ei (--) *enable interrupts*
di (--) *disable interrupts*

Enable or disable interrupts.

Must not be called in **irpt()** and **init()** because these functions are called with interrupts disabled and device selected and must keep interrupts disabled and their device selected.

Note: Disabling interrupts with *di* also selects the device for i/o operations. All i/o opcodes work on the *currently selected device*! The bytecode loader must store the device's actual i/o address with the *di* opcode.

wait (--) *halt CPU & wait for interrupt*

Halt the CPU and wait for an interrupt from this or any other device.

Interrupts must be enabled when *wait* is executed.

wait may resume erroneously.

wait might be used in a block device driver after issuing a command to the drive and waiting for the drive to become ready for i/o, so that during motor-on or seek other interrupts are served.

Note: Assuming that there's a timer interrupt, this will provide a timeout for the *wait* opcode itself. The driver can use opcode *systemtime* to detect a device timeout.

systemtime (-- n) *get low **int16** of system time*

Get the low **int16** word of the current system time which should be roughly in *msec*. Overflows roughly once a minute. Provided for device timeout measurements after opcode *wait*.

timer (--) *call the system timer handler*

A device may provide a timer interrupt which can be used as the system's time base.

If the timer interrupt was enabled during device initialization by a call to **systemtimer()**, then the interrupt handler procedure **irpt()** must call the *timer* opcode once for every timer interrupt. The target system will implement the *timer* opcode to perform all required actions.

If the system provides it's own timer interrupt, then there is no need to implement this opcode.

Code flow:

label (L --) *pseudo opcode: define label **L** here*
jp (L --) *jp to label **L***
jp0 (L n --) *jp to label **L** if **n** is FALSE*
jp1 (L n --) *jp to label **L** if **n** is TRUE*
and0 (L n --) *keep **n** and jp to **L** if **n** is FALSE, else drop **n** & don't jump*
or1 (L n --) *keep **n** and jp to **L** if **n** is TRUE, else drop **n** & don't jump*

Labels are 1-byte numbers following inline after a jump opcode or the label definition opcode. The bytecode loader collects all labels while loading the bytecode of a procedure and remembers their actual position. Finally it updates the addresses in all jump opcodes.

The jump opcodes *jp*, *jp0* and *jp1* are used to construct loops and conditional branches, e.g. for instructions like *IF*, *ELSE*, *WHILE* or *LOOP,* and the pruning operator *'?:'*.

Opcode *and0* is used for pruning boolean operator *AND*.

Opcode *or1* is used for pruning boolean operator *OR*.

Pruning *AND* and *OR* operations only yield only 0 or 1 if their input values are only 0 or 1. Other non-zero 'TRUE' values are left on the stack 'as is'.

switch (L[] n --) *jump to label **L[n]***

The *switch* opcode is followed by a length-prefixed list of labels. Labels are 1-byte numbers and refer to the labels defined with the *label* pseudo opcode.

If *n* is inside the size of the list, then jump to label ***L[n],*** else resume after the *switch* opcode.

call (? n -- ?) *call procedure*

Call procedure with Proc ID ***n***. The procedure ID was defined in the *PROCDEF* chunk. Procedures can only be called when they are already known to the bytecode loader. This means that they must be defined before any caller.

Arguments and return value depend on the called procedure and are indirectly known to the bytecode loader.

ret (? --) *return from procedure*

The return opcode must only occur once at the end of a procedure. When *ret* is executed, the appropriate return value for the current function must be on the stack.

The *ret* opcode indicates the end of code to the bytecode loader.

The bytecode loader must generate code which preserves the return value – if there is one, while it removes all local variables and arguments

from the stack before it returns. Both informations are indirectly known to the bytecode loader from the header info in the *PROCDEF* chunk.

Allocated local variables, i.e. **arrays** and **structs**, must have been already *dispose*d by the program so that *ret* effectively only needs to adjust the data stack pointer.

Notes

Notes on dynamic memory

There are several methods to define ownership of memory. Bytecode has been designed and restricted to be agnostic of the actual method used by a system.

For this reason Bytecode does not support returning arrays or structs from procedures.

Bytecode duplicates and forgets **array** and **struct** pointers without informing the system. It is always assumed that the original *primary* pointer will keep the object allocated and thus any *secondary copy* of the pointer remains valid.

Programmers should clearly distinguish between variables with *primary* pointers and variables with *secondary* pointers. *Primary* pointers are returned by *alloc* and *ival* and when stored in a variable this holds the *primary* pointer. Only *primary* pointer variables must be *dispose*d when destroyed.

Secondary pointers are created by copying a *primary* or a *secondary* pointer; mostly with a variant of *get.p*.

Arguments to procedures must be *secondary* pointers; don't pass an *ival* or an *alloc*ated **array** or **struct** directly; no one will dispose it!

Do not store a *secondary* pointer into a *primary* pointer variable!
Do not store a *primary* pointer into a *secondary* pointer variable!
Do not store a *primary* pointer into a procedure argument variable!

Notes on systems with movable dynamic memory

Most small systems have no memory management unit and cannot use virtual addresses. In order to reclaim and reuse memory they may move memory blocks to merge gaps between them, e.g. in a garbage collection.

Bytecode assumes that pointers to arrays and structs remain valid and don't need to be tracked. The usual approach to solve this, is to use *handles*: pointers into a special block of pointers which point to the actual data. The pointers in this block itself are not moved and so Bytecode can use pointers to those. The system's implementation of the *array* and *struct* opcodes will reflect this.

Two opcodes return a reference to a variable in dynamic memory: *ivar* and *ati*. There's a risk that this pointer becomes invalid before it is consumed by *addgl*, *peekpp*, *set.p* or similar. This can happen, if some other code, which triggers a garbage collection, is executed between the producer and the consumer opcode.

Therefore Bytecode does not allow memory allocation and disposal in the interrupt handler function **Irpt()**. So interrupts are out of the way.

If the system also uses some kind of threads, then it must assert that no thread switching occurs between the producer and the consumer opcode. This can be done by either switching threads only after certain opcodes, e.g. after *branches*, *call* and *ret*, or by switching after all opcodes and skip the dispatcher test only after *ivar* and *ati*.

Switching threads at arbitrary machine code positions is probably not possible in systems with movable dynamic memory, or requires lots of interrupts on and off.

Therefore all opcodes which return a reference to a variable in allocated memory, i.e. *ivar* and *ati*, must be followed by the consumer opcode directly. For this reason all opcodes which read or write to variables in ram always take the reference to the variable as their last argument.

Notes on systems with unusual word size

Most systems will implement **int16** words with 16-bit words. If a system uses an unusual word size then it must mask off the excess bits in the results or operands of some opcodes. This is noted in the description of these opcodes.

If a program creates overflow beyond 16 bits, then programmers must be aware that values on the stack might have excess bits set beyond the 16th bit. But for the most common candidates Bytecode requires masking off these bits, so this should only rarely be a problem.

Notes on using same size for **int8** and **int16**

int8 is defined to be *at least* 8 bits wide. The use of **int8** instead of **int16** in the driver code is merely a hint to the bytecode loader, that 8 bits are sufficient, but the system may use a larger data type for it.

Small 8-bit systems will probably use one 8-bit byte for **int8**, but systems with a 16-bit CPU or larger may want to implement **int8** with the same size as **int16**. This may lead to faster code by avoiding misaligned data access, and padding with 0 when reading data from **int8** variables, and eliminates some of the **int8** opcodes in range *$1E to $2C*.

16-bit systems with 8-bit bytes

A 16-bit system with 8 bits per byte may use *two* bytes for all **int8 variables** same as for **int16**, but will still want to use *one* byte per item in **int8 arrays**.

Now we have a problem with opcodes which return or take a reference to an **int8** value as argument or result. These are opcodes *lget.b*, *gget.b*, *iget.b* and

ati.b (producer) and opcodes *addgl.b* to *mmpeek.b* (consumer). They always occur in pairs of producer/consumer. They are now expected to produce or return a reference to 2-byte data, except for *ati.b*, which will still return a reference to 1-byte data. This must be handled properly:

• Opcodes *ati*, *atiget* and *atiset* are still different for **int8** and **int16**.

• Opcodes for **bytes** *($21 to $2C)* and **words** *($30 to $3B)* can be merged.

• The bytecode loader must detect opcode *ati.b* and use the 1-byte variant of opcodes *addgl.b* to *mmpeek.b* for the following consumer opcode. This case is very unusual, though. It only happens for read-modify-write to **int8** array items.

Systems with 16-bit bytes

Though very unusual, a system may have **bytes** of 16 bits (or larger). Then all variable access opcodes for **bytes** *($1E to $2C)* and **words** *($2D to $3B)* are the same and can be joined.

For an example of a pure 16-bit CPU see http://k1.spdns.de/Develop/Hardware/K1-Computer/K1-CPU/.

Notes on Code Optimization

Bytecode is designed to represent fairly optimal code. Some points for optimizations are left in favour of less opcodes.

In general, performance can be increased by combining frequently occuring code pairs (or triples…) into a new single opcode. Bytecode contains numerous opcodes which actually already combine two functions; e.g. *lget.b* is a combination of *lvar* plus *get.b*.

Examples:

Combine *ival* plus *operator*, e.g. *ival.b* plus *add* to *'addi.b'* (*add immediate*).

Combine *ival* plus *i/o opcode*: the last argument is always the i/o address.

Change Log

1.02 Changed arguments for functions which take blocks from *a, e* to *i, n*.
Added *min*, *msbit* back in, shifted following opcodes.

1.03 Added notes on using same size for **int8** and **int16**.

1.05 Added *not.p*, removed *swapwithvar* and *call_kill*.

1.06 Removed *tor*, *fromr*, *drop* and *drop.p* by integrating them into *ret*.

1.07 Replaced 'subdev' **int** argument in device procs with a **struct** reference.
Added requirement to implicitely add *di* and *ei* for a call to **irpt()**.
Combined chunk 5 and 6: copyright message.
Removed *ati.p*, *atiget.p*, *alloc.p* and *count.p*.

1.08 Switched from *call* proc by name to *call* proc by ID.

1.09 Minor rework of the last rework.

1.10 Revised proc IDs of public procedures.

1.11 Revised argument order of *readi2c* and *writei2c*.

1.12 Changed arguments for functions *gets*, *puts*, *readblocks* and *writeblocks* from *i, n* to *a, e*.

To Do

COMMANDs: Must be one function currently. Should support subroutines.

VIDEO devices

AUDIO devices

Keyboard devices

Pointer devices