The Swimmer Specification

The Swimmer Specification

Table of Contents

Overview ... 1
- 1. Introduction ... 2
- 2. Memory ... 3
  - 2.1. Memory Introduction ... 3
  - 2.2. Virtual Memory ... 3
  - 2.3. Data Sendback ... 4
  - 2.4. Register usage ... 4
- 3. Execution Unit ... 5
- 4. Resources & Packages ... 6
  - 4.1. Resources Introduction ... 6
  - 4.2. Naming ... 6
  - 4.3. Management ... 6
  - 4.4. Loading ... 7
  - 4.5. Packages ... 7
- 5. Native Calls ... 8
  - 5.1. Native Calls Introduction ... 8
  - 5.2. Multicalls ... 8
- 6. Graphics ... 10
- 7. Sound ... 11
- 8. Events & Exceptions ... 12
  - 8.1. Events ... 12
  - 8.2. Exceptions ... 12
- 9. Features ... 13
- 10. Protocol ... 14
- 11. Client startup & flow ... 15

Reference ... 16
- 1. List of IO Registers ... 17
- 2. List of Events ... 19
  - 2.1. `VEC_VBLANK` ... 19
  - 2.2. `VEC_KEYPRESS` ... 19
  - 2.3. `VEC_SCREENCLICK` ... 19
  - 2.4. `VEC_TEXTLINE` ... 19
  - 2.5. `VEC_TIMER` ... 19
  - 2.6. `VEC_RESULT` ... 19
  - 2.7. `VEC_EXCEPTION` ... 19
  - 2.8. `VEC_EXTERNAL` ... 20
  - 2.9. `VEC_NETSTATE` ... 20
- 3. List of Opcodes ... 21
  - 3.1. Zeropage opcodes ... 21
  - 3.2. Miscellaneous Opcodes ... 21
  - 3.3. Memory Opcodes ... 22
  - 3.4. Arithmetic Opcodes ... 23
  - 3.5. Conditional Branch Opcodes ... 25
- 4. List of Network Packets ... 27
  - 4.1. `CallPacket` ... 27
  - 4.2. `EventPacket` ... 27
  - 4.3. `DataPacket` ... 27
  - 4.4. `SendBackPacket` ... 27
  - 4.5. `FeaturePacket` ... 28
  - 4.6. `CryptPacket` ... 28
  - 4.7. `DebugPacket` ... 28

5. List of Client Features 29
6. List of Native Functions 31
6.1. `CALL_NOOP` 31
6.2. `CALL_SEND_EVENT` 31
6.3. `CALL_SEND_USER_EVENT` 31
6.4. `CALL_SEND_DATA` 31
6.5. `CALL_MEMCPY` 31
6.6. `CALL_MEMSET` 31
6.7. `CALL_QSORT` 32
6.8. `CALL_ENTER_LINEMODE` 32
6.9. `CALL_CREATE_RESOURCE` 32
6.10. `CALL_LOAD_RESOURCE` 33
6.11. `CALL_CLEAR_AREA` 33
6.12. `CALL_CREATE_IMAGE` 33
6.13. `CALL_CREATE_IMAGE_FR` 34
6.14. `CALL_SET_RENDER_TARGET` 34
6.15. `CALL_DEFINE_IMAGE_TILING` 34
6.16. `CALL_RENDER_IMAGE` 34
6.17. `CALL_SET_ENCRYPTION` 35
6.18. `CALL_LOAD_PACKAGE` 35
6.19. `CALL_MAP_MEM` 35
6.20. `CALL_DEBUG_PRINT` 35
6.21. `CALL_ALERT` 35
6.22. `CALL_CREATE_SOUND` 36
6.23. `CALL_CREATE_SOUND_FR` 36
6.24. `CALL_PLAY_SOUND` 36
6.25. `CALL_REMOVE_RESOURCES` 36
6.26. `CALL_GET_FEATURE` 37
6.27. `CALL_RESET` 37
6.28. `CALL_GENERATE_RANDOM` 37
6.29. `CALL_SET_MULTICALL` 37
6.30. `CALL_DO_MULTICALL` 38
7. Debugging (obsolete) 39
7.1. Setup 39
7.2. Commands to server 39
7.3. Commands to client 39
7.4. Results from client 41
A. Assembler Examples 43
1. Drawline 44

Overview

1. Introduction

The core of the Swimmer technology is the client, which can also be seen as the *slave* or *terminal* in the system.

The Swimmer Client is a light weight application that can be run on most platforms, but the primary target are portable devices. This Target device is expected to have a display, a network connection and some sort of input system (at least a telephone keyset or a 4-way joypad with action buttons or similar).

The client connects to a server and then runs a normal application loop. Any events (key presses, mouse clicks, timer events etc) are sent to the Virtual Machine to be handled. Initially, most events results in an event packet being sent to the server, so a swimmer application can be entirely located on the server side, with the client only acting as a dumb terminal.

The server controls the client, and can send data into the memory of the virtual machine; graphics, sound or other content goes into data memory, while machine code to be executed goes into program memory.

A typical swimmer application will execute the main bulk of the logic on the server side, while display code and other logic that runs in realtime is implemented in Swimmer Machine Code and sent down to the client.

For instance, in a 2D game where players can move over a giant world, the client renders the background of the current screen, moves the player around depending on key presses, and checks collision. The server decides when it is time to scroll and generates and sends down new data representing the immediate surroundings. From the clients point of view it is a simple one-screen game with a few sprites and tiles to be rendered. On the server side the game can be as complex as needed - for instance implementing a large, multiplayer environment.

NOTE: Most constants defined in this document are only referred to by symbolic values. There actual values are defined in the `swimmer.const` file in the standard Swimmer client source code.

2. Memory

2.1. Memory Introduction

There are six types of memory which can all be written to directly by a swimmer server;

- Data Memory
- Program Memory
- The Vector Table
- IO Memory
- Resource Memory
- Registers

Data Memory is the main chunk of memory. It will contain most of the non-resource data. The first part of the Data Memory is the Zero Page, which is normally used for constants and global variables since it is easily accessible by the Execution Unit, although registers are better in most cases.

Program Memory holds *Swimmer Machine Code*, and can be the target of pointers in the Vector Table.

The *Vector Table* is 256 words that points to code - either through an offset into Program Memory or by referring to a specific *Native Function*. The vector table is used whenever an event occurs or a CALL opcode is executed by the Execution Unit.

IO Memory is a small set of registers that gives access to such things as displaymode, cliparea, timers and network info. It can be seen as a set of memory mapped “hardware” registers.

Resource Memory is not physically mapped memory, but is instead a virtual mapping to the latest created or loaded resource. This allows the server to send the contents of a resource through normal data packets. It also allows the client to access a resource through direct memory access instead of reading it explicitly (similar to the posix `mmap()`-function).

There are also 256 general purpose *Registers* inside the execution unit that is also considered memory. It is used for temporary storage when executing swimmer code, as the arguments for native (and non-native) functions and as global variables that can be set and read by both server and client.

2.2. Virtual Memory

Whenever Data memory is referenced, addresses are translated through a Virtual Memory System. When the client starts up, only two mappings exist - the zero page (initially 256 words) and the IO Memory.

The Virtual Memory works like this:

- The client keeps a list of 256 *banks*. Each bank refers to a Memory section. Each bank also defines that that memory as 8, 16 or 32 bit.
- Whenever a Virtual Address is used, the top 8 bits are taken to be the bank number, and the bottom 24 bits are the offset into the memory the bank points to, counted in units of the bit size of that bank.
- The server can create a new mapping by calling the native function `CALL_MAP_MEM`. When mapping new Data Memory, a new section of the given size is allocated and pointed to by the given bank. For other types of memory, then bank simply refers to the start of that memory after the call.

The client initially maps bank `0x00` to a 32bit Data Memory area of 256 words, and bank `0x60` to IO Memory. This mapping should also be reset after a call to *CALL_RESET*.

If a bank pointing to data memory is remapped (by calling `CALL_MAP_MEM` again with the same bank as argument), the old memory area will be freed.

The bitsize and length are really only considered when mapping data memory, since those attributes are fixed for other types of memory.

2.3. Data Sendback

This is a list of sequences in Data Memory that should be sent back to the server whenever they are changed. This is typically implemented by keeping a copy of all affected memory words and comparing differences and sending them (once each application loop, after events have been handled). A server sets up these sequences through a `CMD_SENDBACK` command.

2.4. Register usage

The 256 Registers are used frequently in the swimmer client. The lower registers are treated differently from the higher ones - mainly; the lower 16 registers are potentially overwritten by event and function calls, and should never contain any global state. R[16] and upwards are only changed by application code. Normally an application allocates global variables starting with R[255] and going down, and uses the lower registers for temporary storage in machine code.

3. Execution Unit

The *Swimmer Execution Unit* - from hereon referred to as the *EU* - is the processor of the virtual machine that is part of the Swimmer Client. It executes Swimmer Code that is present in Program Memory. Each opcode is 32bit in size.

Code in program memory is only executed as a result of an event or a call packet from the server. The client works with a single thread so when the EU is interpreting code the client will not be able to do anything else. If the swimmer client spends too much time executing code (so that the number of opcodes executed reaches the value in `IO_MAXCYCLES`) an exception is generated.

4. Resources & Packages

4.1. Resources Introduction

A Swimmer *Resource* is a persistent peace of data that the client can read into memory or directly create images and sounds from. Resources are created using the `CALL_CREATE_RESOURCE` native call and populated either directly from memory or through subsequent `CMD_WRITEMEM` packets to `RESOURCE_MEM`.

Once a resource has been created, it can be loaded into memory with `CALL_LOAD_RESOURCE`, or used to create an image with `CALL_CREATE_IMAGE_FR`, or a sound using `CALL_CREATE_SOUND_FR`.

The normal way for a server to send and create resources on a client is to first just assume that it is already there and try calling `CALL_CREATE_IMAGE_FR` for instance. If the call fails, the server creates and sends the resource, and tries again.

All resources have a type, which is a 32bit identifier stored with the resource (normally as the first 4 bytes), telling the client how to handle it.

4.2. Naming

Resources are referred to using a combination a textual name and an md5 checksum, either or both of which can be specified. It is usually best to create resources using both, and load them using only the checksum. That way you can make sure old resources are discarded and replaced by updated resources from the server.

The textual name can then be used when removing resources; it is for instance possible to remove all resources with the same name *except* for the one with the given MD5 - this is how you remove all old versions of a resource but keep the current one.

Resources that contains volatile data (settings, highscores) are normally only created and referred to by name.

4.3. Management

Resources are application specific, so they are normally be stored as files in a directory named after the current *Application ID*. They could be stored in any way though, as long as the resources are kept separate.

Recommended naming: `<resource directory>/<appid>/<md5>=<name>`

An exception to this rule can be made when reading a resource using the MD5 checksum. If another application has a resource with the same checksum they will by definition have the same content so there is no security issues with reading the file, even though it resides in another applications storage.

If persistent memory is running low, a client is expected to remove the resources that has not been accessed for the longest time, but *only those* who have checksums (since a checksum indicates a non volatile resource that most likely exists on the server and can be sent again).

If runtime memory is running low, a client could use a resource cache and flush out unused resources and load them again when used.

4.4. Loading

A resource can be loaded directly into memory, whole or in parts, using `CALL_LOAD_RESOURCE`. Plain binary (RTYPE_RAW) resources are loaded directly without conversion (endianess are assumed to match the client).

If a resource has the type `RTYPE_GZIP` it is assumed to be gzip compressed, raw 8 bit data. The file offset and size refers to the unpacked file in this case.

If the client supports stream formats, loading such a resource into memory should also decompress/convert this data to raw. This lets the client, for instance, use large MP3 resources and stream from them.

(Note that if you instead create a *sound resource* from an MP3, the data is normally *not* decompressed but played directly as MP3).

4.5. Packages

A Package is a special type of resource that contains a stream of network packets. It can be used by the server to cache data that does not fit into a general resource (Code in program memory, sequences of calls etc). It is also the method used to create offline applications for Swimmer.

A package has the resource type `RTYPE_PACKAGE` and is normally only used by the `CALL_LOAD_PACKAGE` native call. The effect is exactly like the packages where sent from the server (except all packages are handled at once, no code or rendering is allowed to take place in between).

5. Native Calls

5.1. Native Calls Introduction

Native calls are the way you perform system specific functions on a client, such as draw on the screen, play sounds, allocate memory and send network data. There are also native calls for functions that could be implemented using machine code, but is much faster as a native function (such as memset or memcopy).

A native function can be called directly from the server by sending a CallPacket, or from machine code by placing the arguments into the corresponding registers and executing the CALL opcode.

In the case of the CALL opcode, the call is not taken directly; instead the given numeric argument is used too look up the 32bit value at that offset in the vector table. If this value is < `0x7FFFFFFF` it is taken as an offset into program memory where execution should continue, but if the high bit is set, the lower bits indicate the number of the native function that should be called.

`R[0]` is used as return value for the functions that returns a value. All other registers are left unchanged. For functions returning no value, *all* registers are left unchanged.

Some functions generates results - they do so by generating a `VEC_RESULT` event with the return code in `R[2]` and the key argument (an argument the identifies the original call, like the image number) in R[3]. R[0] to R[3] are saved before inserting arguments, and restored after the event was handled.

Some functions that are not necessary can be left unimplemented on certain clients (like `CALL_DEBUG_PRINT` on a device with no logging capabilities) - in that case it should be treated like a noop and silently do nothing.

A call to an unknown function should generate a RC_ERR_CALL_NOT_FOUND result. This way a server can test for certain functions that may or may not be implemented.

5.2. Multicalls

Multicalls are a special type of native calls - they let you call any native call (referred to as the target call) a multiple number of times with one single call.

Only native calls with 6 or less arguments are supported. R[6] contains either the target call or the call count.

A multicall is first set up using the `CALL_SET_MULTICALL` and then performed using the `CALL_DO_MULTICALL`. For `CALL_SET_MULTICALL` you specify the actual target call, how the arguments should be interpreted, and how they should be modified after each successive target call. For `CALL_SET_MULTICALL` you specify the initial values of the arguments, and the number of times you want to call the target call.

Each argument is incremented or decremented by a specific value for each target call.

Each argument can also either be used directly (a normal constant) or indirectly. In the latter case, the argument is a pointer into data memory where the actual value for the call resides.

For instance, using an increment that is a multiple of the screen width as the Y value of a rendering call lets you render a horizontal stretch (of images for instance). Or combining an indirect argument with an increment of 1 lets you use a list of arguments.

How multicalls are performed

When `CALL_SET_MULTICALL` is called, the first 7 arguments are saved into internal registers (referred to as *MC[0]* to *MC[5]* and *TargetCall*, respectively). `IO_XDIVIDER` and `IO_YDIVIDER` are saved into *MCX* and *MXY*.

When `CALL_DO_MULTICALL` is called, `TargetCall` is executed `R[6]` number of times, in this way;

```
// S24(x) = Sign extend from 24bit

count = R[6]

for(i=0; i<6; i++)
{
  if(MC[i] & MC_INDIRECT)
  {
    B[i] = R[i] & 0xff000000
    S[i] = R[i] & 0x00ffffff
  }
  else
    S[i] = R[i]
}

while(count--)
{
  for(i=0; i<6; i++)
  {
    divider = 1
    if(MC[i] & MC_XDIV)
      divider *= MCX
    if(MC[i] & MC_YDIV)
      divider *= MCY

    if(MC[i] & MC_INDIRECT)
      R[i] = GetWord(B[i] | (S[i] / divider))
    else
      R[i] = S[i] / divider
  }

  Call(TargetCall)

  for(i=0; i<6; i++)
      S[i] += S24(MC[i] & 0xffffff)
}
```

6. Graphics

The only way to display graphics on a client is by rendering *Images* or rectangles.

...

7. Sound

...

8. Events & Exceptions

8.1. Events

The swimmer virtual machine generates *Events* whenever something occurs that may be of interest to the application, such as a key press or the timer reaching a trigger value.

8.2. Exceptions

The Swimmer virtual machine generates *Exceptions* as a response to special conditions (normally errors) that can occur, such as illegal accesses to memory or trying to render an image that does not exist.

This is what happens when an exception is triggered;

- If `IO_EXCP_EX` is not zero, the exception becomes *Fatal*, normally causing the application to exit.
- The current state is stored into the `IO_EXCP` registers (the exception number, the current program counter, the current native call (if any), and the memory adress accessed (if any).
- The stack pointer is set to zero.
- If `VECTORS[VEC_EXCEPTION]` points to a native call, r0 to r1 is set to `VEC_EXCEPTION`, `IO_EXCP_EX`, `IO_EXCP_PC` and `IO_EXCP_CALL` respectivly.
- A `VEC_EXCEPTION` event is triggered, causing the code or native call referred to by `VECTORS[VEC_EXCEPTION]` to be executed.

At this point, the application loop continues, and EU code is not allowed to run until `IO_EXCP_EX` is zero. Note that this condition can not be present during the `VEC_EXCEPTION` handling, because then any exception handling code would not be allowed to run.

9. Features

Whenever a client connects, the first thing it does is send a `FeaturePacket`, which will tell the server the features and capabilities of the client, so that it can adapt its content to it.

...

10. Protocol

The Swimmer Protocol is what defines how data is sent between the client and server over the network. It is based on a small number of Packets, each with the same 32bit header.

...

11. Client startup & flow

This section describes the general startup procedure and main application loop of a Swimmer Client.

Reference

1. List of IO Registers

Name	Size	Description
`IO_APPID`	128bit	Unique application ID. May only be written by the server. Should preferably be completely random number generated once and for all on the server side, and kept static throughout the application's lifetime. The client normally uses this ID to tag created resources, so that two resources with the same name will not collide, as long as they are created by different applications.
`IO_TIMER`	32bit	A microsecond timer that can be both read and written. The server will typically write to it when it is trying to synchronize all clients to the same "time space". The timer value is also automatically sent along with Event packets to the server.
`IO_TIMER_TRIGGER`	32bit	When `IO_TIMER` == `IO_TTRIGGER`, a `TIMER_EVENT` will be generated on the client.
`IO_SERVERIP`	32bit	The server to connect to, the next time a connection is made. Redirection works by the server sending a new IP to this memory and then simply disconnecting the client. A value of zero indicates that connection should be made to the default or initial server (specified on the command line or in a configuration file for instance).
`IO_SERVERPORT`	32bit	The port for the connection, works like the IP.
`IO_CYCLES`	32bit	Counts up for each cycle executed. Used for benchmarking. May be unavailable in non-debug versions of the client, for perfomance reasons.
`IO_MAXCYCLES`	32bit	Maximum number of cycles the execution unit is allowed to run, before returning back to the client. If this value is exceeded, an `EXCP_MAX_CYCLES` exception is generated.
`IO_GAMEPAD`	32bit	Indicates which of the basic gamepad keys are pressed. At least `GPAD_RIGHT`, `GPAD_LEFT`, `GPAD_UP`, `GPAD_DOWN` and `GPAD_ACTION` are available.
`IO_POINTERX`	32bit	The current X coordinate of the pointer device (touch screen or mouse).
`IO_POINTERY`	32bit	The current Y coordinate of the pointer device.
`IO_POINTERBTN`	32bit	The button state of the pointer device. Lowest bit is first (left) button etc.
`IO_DISPLAYMODE`	32bit	Read or write the display mode; `0` means single buffered and `1` means double buffered. Single buffered is the mode the client uses per default, and mean either that it uses a single screen buffer and all graphic operation becomes instantly visible, or it uses two buffers where the back buffer is copied to front buffer as necessary. In both cases no special care needs to be taken when rendering, the client can rely on that graphcis stays on the screen between `VEC_VBLANK` events.

Name	Size	Description
		Double buffering means the client may use two buffers and flip between them, for maximum efficiency. This means the client code needs to redraw the entire screen each frame (each time `VEC_VBLANK` is called).
`IO_CLIPX0`	32bit	Clip coordinate for image rendering. Defaults to 0.
`IO_CLIPY0`	32bit	Clip coordinate for image rendering. Defaults to 0.
`IO_CLIPX1`	32bit	Clip coordinate for image rendering. Defaults to the screen width.
`IO_CLIPY0`	32bit	Clip coordinate for image rendering. Defaults to the screen height.
`IO_NETSTATE`	32bit	Current state of the Network. The following values can be read/written: `NS_DISCONNECTED` - The client is disconnected and will not try to connect. `NS_CONNECTING` - The client is disconnected but will try to connect to `IO_SERVERIP`:`IO_SERVERPORT`. `NS_CONNECTED` - Read only, the client is connected. `NS_RESET` - Write only, will cause the client to reset all memory and send the feature packet again, just like a reconnect except it will not close the socket. The actual network state should not change. Note also that the feature packet is only sent if state was `NS_CONNECTED`. `NS_QUIT` - Write only, will cause the client to quit.
`IO_BYTESSENT`	32bit	Total number of bytes sent since client started.
`IO_BYTESRECVD`	32bit	Total number of bytes received since client started.
`IO_SRAM`	256bit	A read/write area in IO ram that is not cleared when disconnecting/ reconnecting. Typically used to send information to a new server when redirecting.
`IO_LATENCY`	32bit	The latency (round trip time over the network, to the server and back) in milliseconds.
`IO_EXCP_EX`	32bit	Set to zero during normal operation, and to an exception value when an exception has occurred - the execution unit will not run code when this is non-zero.
`IO_EXCP_CALL`	32bit	The native call in which the exception occurred, if any.
`IO_EXCP_ADR`	32bit	The memory address of the last exception - only valid for memory read/write exceptions.
`IO_EXCP_PC`	32bit	The current program counter if the EU was running.
`IO_XDIVIDER`	32bit	This value is used to divide the X coordinate of all rendering native calls. It defaults to 1.
`IO_YDIVIDER`	32bit	This value is used to divide the Y coordinate of all rendering native calls. It defaults to 1.

2. List of Events

R[0] = Always contains the number of the event.

2.1. VEC_VBLANK

This event is sent once every application loop, which on supported hardware should be once every time the screen refreshes, or around 30 times per second. It is where you normally place your rendering code.

2.2. VEC_KEYPRESS

This event is sent as soon as the user presses or releases any button on the device.

R[1] = Keycode (UNICODE between 0x20 and 0xFFFF or a special key if 0x10000 and above)
R[2] = Above 0 if a key was pressed, 0 if released
R[3] = Device Number (0 for keyboard, 1 for first gamepad and so on)

2.3. VEC_SCREENCLICK

This event is generated when a mouse button, touch screen or other pointing device is pressed or released.

R[1] = X position (in pixels)
R[2] = Y position (in pixels)
R[3] = Above zero If key/pen is down (pressure), 0 if up

2.4. VEC_TEXTLINE

This event is triggered when a line of text has been entered by the user.

R[1] = Pointer to entered text (same value as given when `VEC_ENTER_TEXTMODE` was called).

2.5. VEC_TIMER

This event is triggered when the current timer value equals the trigger value. To make the event occur continuously, the triggered code should read out the trigger value (`IO_TIMER_TRIGGER`) add the appropriate value and write it back.

2.6. VEC_RESULT

This event is sent to inform about the result of a previous call, and exists so that the server can know the result of important calls.

R[1] = Function (Call vector) that generated the result.
R[2] = Return code
R[3] = Key argument (Meant to identify which specific call this is in response to, if several calls where made to the same function.

2.7. VEC_EXCEPTION

This may not be considered a real event - it is triggered when the Virtual Machine generates an exception.

2.8. VEC_EXTERNAL

This event is triggered when special hardware wants to signal that something has happened. It is intended to be used for when a photo is taken, an SMS arrives, the phone rings etc.

2.9. VEC_NETSTATE

This event is triggered when the network changes state - either disconnecting or connecting to a server. If this event is overridden, and R[1] = `NS_CONNECTING`, the code should call `CALL_RESET` with an argument of `0xFFFF` to emulate the normal behaviour of the client.

3. List of Opcodes

This table shows the basic opcodes. In the pseudo code, `n3`, `n2`, `n1` and n0 are the 4 bytes of the opcode (with `n3` being most significant) and `b1` and `b0` are the high and low 16bit word. `R[x]` is the contents of register x, `MEM[x]` is the word at data memory address x and `VEC[x]` is the x:th call vector. Also note: PC is the value of the program counter after the instruction has been decoded.

3.1. Zeropage opcodes

Name	Opcode	Description	Pseudo code
`OP_LOADZ`	**LOAD** [*adr*], R*d*	Load direct from memory in bank zero (zeropage)	`R[n2] := MEM[b0]`
`OP_STOREZ`	**STORE** [*adr*], R*s*	Store direct to memory in bank zero (zeropage)	`MEM[b0] := R[n2]`
`OP_ADDZ`	**ADD** R*d*, [*adr*]	Add zeropage to register	`R[n2] += MEM[b0]`
`OP_ANDZ`	**AND** R*d*, [*adr*]	And zeropage to register	`R[n2] &= MEM[b0]`
`OP_ORZ`	**OR** R*d*, [*adr*]	Or zeropage to register	`R[n2] |= MEM[b0]`
`OP_XORZ`	**XOR** R*d*, [*adr*]	Exclusive Or zeropage to register	`R[n2] ^= MEM[b0]`
`OP_MULZ`	**MUL** R*d*, [*adr*]	Multiply register by zeropage	`R[n2] *= MEM[b0]`
`OP_DIVZ`	**DIV** R*d*, [*adr*]	Divide register by zeropage	`R[n2] /= MEM[b0]`

3.2. Miscellaneous Opcodes

Name	Opcode	Description	Pseudo code
`OP_BRAR`	**BRA** [R*s*]	Branch via register	`PC := R[n0]`

`OP_BSRR`	**BSR** [R*s*]	Branch to subroutine via register	`*SP++ := PC`
			`PC := R[n0]`

`OP_RET`	**RET**	Return from CALL or BSR	`if(SP == STACK)`
			`  return()`
			`else`
			`  PC = *(--SP)`

`OP_BRAF`	**BRA** *adr*	Non conditional branch forward	`PC += b0`
`OP_BRAB`	**BRA** *adr*	Non conditional branch backward	`PC -= b0`

`OP_BSRF`	**BSR** *adr*	Branch to subroutine forward	`*SP++ := PC`
			`PC += b0`

`OP_BSRB`	**BSR** *adr*	Branch to subroutine backward	`*SP++ := PC`
			`PC -= b0`

`OP_BRAZ`	**BRA** *adr*	Branch absolute from zeropage	`PC := MEM[b0]`

`OP_CALL`	**CALL** c	Software Interrupt (Native call)	`*SP++ := PC`
			`call(VEC[b0])`
			`PC := *(--SP)`

`OP_PUSH`	**PUSH** R*s*, R*s*	Push registers to the stack	`for(i=n1 to n0) SP[i] := R[i]`
			`SP += (n1-n0+1)`

Name	Opcode	Description	Pseudo code

`OP_POP`	**POP** R*s*, R*s*	Pop registers from the stack	`SP =- (n1-n0+1)`
			`for(i=n1 to n0) R[i] := SP[i]`

`OP_TRUNC`	**TRUNC** R*s*, *c*, *c*	Truncate a value between a low and a high value.	`if(R[n2] < PC[n1])`
			`  R[n2] := PC[n1]`
			`else`
			`  if(R[n2] > PC[n0])`
			`    R[n2] := PC[n0]`

`OP_DBGTF`	**DBGT** R*d*, *c*, *adr*	Decrement and branch if greater than zero.	`R[n2] -= PC[n1]`
			`if(R[n2] > 0)`
			`  PC += n0`

`OP_DBGTB`	**DBGT** R*d*, *c*, *adr*	Decrement and branch if greater than zero.	`R[n2] -= PC[n1]`
			`if(R[n2] > 0)`
			`  PC -= n0`

`OP_BLEDF`	**BLED** R*d*, *c*, *adr*	Branch if lower or equal to zero, otherwise decrement.	`if(R[n2] <= 0)`
			`  PC += n0`
			`else`
			`  R[n2] -= PC[n1]`

`OP_BLEDB`	**BLED** R*d*, *c*, *adr*	Branch if lower or equal to zero, otherwise decrement.	`if(R[n2] <= 0)`
			`  PC -= n0`
			`else`
			`  R[n2] -= PC[n1]`

`OP_SWAP`	**SWAP** R*d*, R*d*	Swap the contents of two registers	`TMP := R[n1]`
			`R[n1] := R[n0]`
			`R[n0] := TMP`

`OP_MIN`	**MIN** R*d*, R*s*, R*s*	Get the lowest of two registers	`if(R[n1] < R[n0])`
			`  R[n2] := R[n1]`
			`else`
			`  R[n2] := R[n0]`

`OP_MAX`	**MAX** R*d*, R*s*, R*s*	Get the highest of two registers	`if(R[n1] > R[n0])`
			`  R[n2] := R[n1]`
			`else`
			`  R[n2] := R[n0]`

3.3. Memory Opcodes

Name	Opcode	Description	Pseudo code

`OP_LOAD`	**LOAD** [R*s*], R*d*, R*d*	Load multiple words indirect from memory	`for(i=n1 to n0)`
			`  R[i] := MEM[i-n1+R[n2]]`

`OP_STORE`	**STORE** [R*s*], R*s*, R*s*	Store multiple words indirect to memory	`for(i=n1 to n0)`
			`  MEM[i-n1+R[n2]] := R[i]`

`OP_LOADP`	**LOAD** [R*s*]+, R*d*, R*d*	Load multiple words indirect from memory, increase after	`for(i = n1 to n0)`
			`  R[i] := MEM[i-n1+R[n2]]`
			`R[n2] += (n0-n1+1)`

`OP_STOREP`	**STORE** [R*s*]+, R*s*, R*s*	Store multiple words indirect to memory, increase after	`for(i=n1 to n0)`
			`  MEM[i-n1+R[n2]] := R[i]`
			`R[n2] += (n0-n1+1)`

Name	Opcode	Description	Pseudo code

`OP_LOAD2B`	**LOAD2B** [R*s*], R*d*	Load 2 bytes as 16bit word	`R[n1] :=`
			`  (MEM[n0] << 8) |`
			`  (MEM[n0+1] & 0xFF)`

`OP_LOAD4B`	**LOAD4B** [R*s*], R*d*	Load 4 bytes as 32bit word	`R[n1] :=`
			`  (MEM[n0] << 24) |`
			`  (MEM[n0+1] & 0xFF << 16) |`
			`  (MEM[n0+2] << 8 & 0xFF) |`
			`  (MEM[n0+3] & 0xFF)`

`OP_LOAD2W`	**LOAD2W** [R*s*], R*d*	Load 2 16bit words as 32bit word	`R[n1] :=`
			`  (MEM[n0] << 16) |`
			`  (MEM[n0+1] & 0xFFFF)`

`OP_LOAD2BP`	**LOAD2B** [R*s*]+, R*d*	Load 2 bytes as 16bit word, increase after	`R[n1] :=`
			`  (MEM[n0] << 8) |`
			`  (MEM[n0+1] & 0xFF)`
			`R[n1] += 2`

`OP_LOAD4BP`	**LOAD4B** [R*s*]+, R*d*	Load 4 bytes as 32bit word, increase after	`R[n1] :=`
			`  (MEM[n0] << 24) |`
			`  (MEM[n0+1] & 0xFF << 16) |`
			`  (MEM[n0+2] << 8 & 0xFF) |`
			`  (MEM[n0+3] & 0xFF)`
			`R[n1] += 4`

`OP_LOAD2WP`	**LOAD2W** [R*s*]+, R*d*	Load 2 16bit words as 32bit word, increase after	`R[n1] :=`
			`  (MEM[n0] << 16) |`
			`  (MEM[n0+1] & 0xFFFF)`
			`R[n1] += 2`

`OP_LDOFFS`	**LOAD** [R*s*, *c*], R*d*	Load word from register with signed constant offset	`if(n1 > 0x7F)`
			`  R[n0] := MEM[R[n2] - 0x100 + n1]`
			`else`
			`  R[n0] := MEM[R[n2] + n1]`

`OP_STOFFS`	**STORE** [R*s*, *c*], R*s*	Store word from register with signed constant offset	`if(n1 > 0x7F)`
			`  MEM[R[n2] - 0x100 + n1] := R[n0]`
			`else`
			`  MEM[R[n2] + n1] := R[n0]`

`OP_LDOFFSR`	**LOAD** [R*s*, R*s*], R*d*	Load word from register with offset register	`R[n0] := MEM[R[n2] + R[n1]]`
`OP_STOFFSR`	**STORE** [R*s*, R*s*], R*s*	Store word from register with offset register	`MEM[R[n2] + R[n1]] := R[n0]`

3.4. Arithmetic Opcodes

Name	Opcode	Description	Pseudo code

`OP_MOVE3C`	**MOVE** R*d*, R*d*, *c*	Write constants to a register sequence	`for(i=n2 to n1)`
			`  R[i] := PC[n0-n2+i]`

`OP_ADD3C`	**ADD** R*d*, R*s*, *c*	Add register and constant to another register	`R[n2] := R[n1] + PC[n0]`
`OP_AND3C`	**AND** R*d*, R*s*, *c*	And register with constant to another register	`R[n2] := R[n1] & PC[n0]`

Name	Opcode	Description	Pseudo code
`OP_OR3C`	**OR** R*d*, R*s*, *c*	Or register with constant to another register	`R[n2] := R[n1] | PC[n0]`
`OP_XOR3C`	**XOR** R*d*, R*s*, *c*	Exclusive Or register with constant to another register	`R[n2] := R[n1] ^ PC[n0]`
`OP_MUL3C`	**MUL** R*d*, R*s*, *c*	Multiply register by constant to another register	`R[n2] := R[n1] * PC[n0]`
`OP_DIV3C`	**DIV** R*d*, R*s*, *c*	Divide register by constant to another register	`R[n2] := R[n1] / PC[n0]`
`OP_RSUB2C`	**SUB** R*d*, *c*, R*s*	Subtract register from constant to another register	`R[n2] := PC[n0] - R[n1]`
`OP_MOD3C`	**MOD** R*d*, R*s*, *c*	Modulo (Division reminder) register by constant.	`R[n2] := R[n1] % PC[n0]`
`OP_ADD3`	**ADD** R*d*, R*s*, R*s*	Add two registers to a third register	`R[n2] := R[n1] + R[n0]`
`OP_AND3`	**AND** R*d*, R*s*, R*s*	And two registers to a third register	`R[n2] := R[n1] |&R[n0]`
`OP_OR3`	**OR** R*d*, R*s*, R*s*	Or two registers to a third register	`R[n2] := R[n1] | R[n0]`
`OP_MUL3`	**MUL** R*d*, R*s*, R*s*	Multiply two registers to a third register	`R[n2] := R[n1] * R[n0]`
`OP_DIV3`	**DIV** R*d*, R*s*, R*s*	Divide two registers to a third register	`R[n2] := R[n1] / R[n0]`
`OP_SUB3`	**SUB** R*d*, R*s*, R*s*	Subtract two registers to a third register	`R[n2] := R[n1] - R[n0]`
`OP_MOD3`	**MOD** R*d*, R*s*, R*s*	Modulo two registers to a third register	`R[n2] := R[n1] % R[n0]`
`OP_LSHIFT3`	**ASL** R*d*, R*s*, R*s*	Shift a register by another register to a third register	`R[n2] := R[n1] << R[n0]`
`OP_RSHIFT3`	**ASR** R*d*, R*s*, R*s*	Shift a register by another register to a third register	`R[n2] := R[n1] >> R[n0]`
`OP_FIXMUL`	**FMUL** R*d*, R*s*, *c*	Fixed point multiply; Multiply and right shift by constant	`R[n2] := (R[n2]*PC[n1])>>n0`
`OP_FIXMULR`	**FMUL** R*d*, R*s*, R*s*	Fixed point multiply; Multiply and right shift by register	`R[n2] := (R[n2]*R[n1])>>n0`
`OP_RDIV3C`	**DIV** R*d*, *c*, R*s*	Divide a constant by a register	`R[n2] := PC[n0] / R[n1]`
`OP_RMOD3C`	**MOD** R*d*, *c*, R*s*	Modulo a constant by a register	`R[n2] := PC[n0] % R[n1]`
`OP_MADD3`	**MADD** R*d*, R*s*, R*s*	Multiply and Add; Add the product of two registers to a third register	`R[n2] += (R[n1] * R[n0])`
`OP_MADD3C`	**MADD** R*d*, R*s*, *c*	Multiply and Add; Add the product of a register and a constant to a third register	`R[n2] += (R[n1] * PC[n0])`

3.5. Conditional Branch Opcodes

Name	Opcode	Description	Pseudo code
`OP_JEQF`	**BEQ** R*d*, *c*, *adr*	Branch if register is equal to constant.	`if(R[n2] == PC[n1])` `PC += n0`
`OP_JEQB`	**BEQ** R*d*, *c*, *adr*	Branch if register is equal to constant.	`if(R[n2] == PC[n1])` `PC -= n0`
`OP_JNEF`	**BNE** R*d*, *c*, *adr*	Branch if register is not equal to constant.	`if(R[n2] != PC[n1])` `PC += n0`
`OP_JNEB`	**BNE** R*d*, *c*, *adr*	Branch if register is not equal to constant.	`if(R[n2] != PC[n1])` `PC -= n0`
`OP_JLTF`	**BLT** R*d*, *c*, *adr*	Branch if register is less than constant.	`if(R[n2] < PC[n1])` `PC += n0`
`OP_JLTB`	**BLT** R*d*, *c*, *adr*	Branch if register is less than constant.	`if(R[n2] < PC[n1])` `PC -= n0`
`OP_JGEF`	**BGE** R*d*, *c*, *adr*	Branch if register is greater than or equal to constant.	`if(R[n2] >= PC[n1])` `PC += n0`
`OP_JGEB`	**BGE** R*d*, *c*, *adr*	Branch if register is greater than or equal to constant.	`if(R[n2] >= PC[n1])` `PC -= n0`
`OP_JLEF`	**BLE** R*d*, *c*, *adr*	Branch if register is less than or equal to constant.	`if(R[n2] <= PC[n1])` `PC += n0`
`OP_JLEB`	**BLE** R*d*, *c*, *adr*	Branch if register is less than or equal to constant.	`if(R[n2] <= PC[n1])` `PC -= n0`
`OP_JGTF`	**BGT** R*d*, *c*, *adr*	Branch if register is greater than constant.	`if(R[n2] > PC[n1])` `PC += n0`
`OP_JGTB`	**BGT** R*d*, *c*, *adr*	Branch if register is greater than constant.	`if(R[n2] > PC[n1])` `PC -= n0`
`OP_JMCF`	**BMC** R*d*, *c*, *adr*	Branch if register anded with constant is zero.	`if(!(R[n2] & PC[n1]))` `PC += n0`
`OP_JMCB`	**BMC** R*d*, *c*, *adr*	Branch if register anded with constant is zero.	`if(!(R[n2] & PC[n1]))` `PC -= n0`
`OP_JMSF`	**BMS** R*d*, *c*, *adr*	Branch if register anded with constant is not zero.	`if(R[n2] & PC[n1])` `PC += n0`
`OP_JMSB`	**BMS** R*d*, *c*, *adr*	Branch if register anded with constant is not zero.	`if(R[n2] & PC[n1])` `PC -= n0`
`OP_JEQFR`	**BEQ** R*d*, *c*, *adr*	Branch if register is equal to register.	`if(R[n2] == R[n1])` `PC += n0`
`OP_JEQBR`	**BEQ** R*d*, *c*, *adr*	Branch if register is equal to register.	`if(R[n2] == R[n1])` `PC -= n0`
`OP_JNEFR`	**BNE** R*d*, *c*, *adr*	Branch if register is not equal to register.	`if(R[n2] != R[n1])` `PC += n0`
`OP_JNEBR`	**BNE** R*d*, *c*, *adr*	Branch if register is not equal to register.	`if(R[n2] != R[n1])` `PC -= n0`
`OP_JLTFR`	**BLT** R*d*, *c*, *adr*	Branch if register is less than register.	`if(R[n2] < R[n1])` `PC += n0`

Name	Opcode	Description	Pseudo code

`OP_JLTBR`	**BLT** R*d*, *c*, *adr*	Branch if register is less than register.	`if(R[n2] < R[n1])`
			`  PC -= n0`

`OP_JGEFR`	**BGE** R*d*, *c*, *adr*	Branch if register is greater than or equal to register.	`if(R[n2] >= R[n1])`
			`  PC += n0`

`OP_JGEBR`	**BGE** R*d*, *c*, *adr*	Branch if register is greater than or equal to register.	`if(R[n2] >= R[n1])`
			`  PC -= n0`

`OP_JLEFR`	**BLE** R*d*, *c*, *adr*	Branch if register is less than or equal to register.	`if(R[n2] <= R[n1])`
			`  PC += n0`

`OP_JLEBR`	**BLE** R*d*, *c*, *adr*	Branch if register is less than or equal to register.	`if(R[n2] <= R[n1])`
			`  PC -= n0`

`OP_JGTFR`	**BGT** R*d*, *c*, *adr*	Branch if register is greater than register.	`if(R[n2] > R[n1])`
			`  PC += n0`

`OP_JGTBR`	**BGT** R*d*, *c*, *adr*	Branch if register is greater than register.	`if(R[n2] > R[n1])`
			`  PC -= n0`

`OP_JMCFR`	**BMC** R*d*, *c*, *adr*	Branch if register anded with register is zero.	`if(!(R[n2] & R[n1]))`
			`  PC += n0`

`OP_JMCBR`	**BMC** R*d*, *c*, *adr*	Branch if register anded with register is zero.	`if(!(R[n2] & R[n1]))`
			`  PC -= n0`

`OP_JMSFR`	**BMS** R*d*, *c*, *adr*	Branch if register anded with register is not zero.	`if(R[n2] & R[n1])`
			`  PC += n0`

`OP_JMSBR`	**BMS** R*d*, *c*, *adr*	Branch if register anded with register is not zero.	`if(R[n2] & R[n1])`
			`  PC -= n0`

4. List of Network Packets

The Swimmer Protocol is used to send data between client and server. It is based on packets of arbitrary sizes. Each packet has the same 32bit header. The size field is in bytes and includes the header size. The size field is big-endian but all the data following is always in the same endian as the client.

4.1. `CallPacket`

Send by server to make client call a function. Can contain up to 12 arguments, each placed in the corresponding register.

Field	Size	Description
CMD	8bit	The constant `CMD_CALL`
PARAM	8bit	Unused
SIZE	16bit	Total packet size in bytes
Callno	32bit	Call reference - either an offset into program memory or a native function
Arg0...	32bit	All arguments to function, placed in `R[0]`, `R[1]` etc

4.2. `EventPacket`

Sent by client to server from the native call `CALL_SEND_EVENT` or `CALL_SEND_USER_EVENT`. Can also be sent from the server to trigger an event on the client.

Field	Size	Description
CMD	8bit	The constant `CMD_EVENT`
PARAM	8bit	The event number
SIZE	16bit	Total packet size in bytes
Arg0...	32bit	All arguments to event. Not present if SIZE = 4

4.3. `DataPacket`

Send by server to place data into client memory, or by client when a memory area marked in the sendback list has changed.

Field	Size	Description
CMD	8bit	The constant `CMD_DATA`
PARAM	8bit	The target memory type. Can be `DATA_MEM`, `PROGRAM_MEM`, `VECTOR_MEM`, `IO_MEM`, `RESOURCE_MEM` or `REGISTER_MEM`.
SIZE	16bit	Total packet size in bytes
Offset	32bit	Offset into client memory (in words) where data should be copied
Word0...	32bit	Data words...

4.4. `SendBackPacket`

Send by server to set up *send back* of certain memory areas. Areas marked for send back will trigger a DataPacket being sent to the server when they are changed.

Field	Size	Description
CMD	8bit	The constant `CMD_SENDBACK`
PARAM	8bit	`MATTR_SENDBACK_ADD` or `MATTR_SENDBACK_REMOVE`
SIZE	16bit	Total packet size in bytes
Offset	32bit	Offset into client memory (in 32bit words) to affected memory area
Length	32bit	The size of the memory area

4.5. `FeaturePacket`

Send by client directly after connecting, to inform server of its capabilities.

Field	Size	Description
CMD	8bit	The constant `CMD_FEATURES`
PARAM	8bit	Unused
SIZE	16bit	Total packet size in bytes
Magic	32bit	The constant 0x12345678 - used to detect endianness
Version	32bit	Client version. Currently `0x00000001`
Data0...	8bit	List of features (explained below)

4.6. `CryptPacket`

All packets are wrapped in this packet during an encryption session.

Field	Size	Description
CMD	8bit	The constant `CMD_CRYPT`
PARAM	8bit	Unused
SIZE	16bit	Total packet size in bytes
Encrypted Data...		Encrypted packet

4.7. `DebugPacket`

Sent to initiate and acknowledge debugging, and for all debug commands after that.

Field	Size	Description
CMD	8bit	The constant `CMD_DEBUG`
PARAM	8bit	Unused
SIZE	16bit	Total packet size in bytes
Id	32bit	Identifies the debugging client

5. List of Client Features

Features without arguments

Feature	Description
FV_HAS_PC_KEYBOARD	The client has a PC-like keyboard.
FV_HAS_TOUCHSCREEN	The client has a touch screen.
FV_HAS_MOUSE	The client has a mouse.
FV_HAS_VIRTUAL_KEYBOARD	A physical keyboard is not present, but the client can generate text by other mean (Using a phonepad or a touch screen for instance).
FV_HAS_PHONEPAD	The client has a numeric keypad laid out like a phone.
FV_HAS_NUMPAD	The client has a numeric keypad like the one on a PC keyboard (mutually exclusive with FV_HAS_PHONEPAD).
FV_HAS_SIMPLE_JOYSTICK	The client has some sort of directional device and an action button. A simple joystick does not need to generate events.
FV_HAS_GAMEPAD	The client has a real joystick device, that can generate events.
FV_HAS_SCREEN	The client has a screen.
FV_HAS_EU	The client has an execution unit (can run Swimmer Code).
FV_HAS_ENCRYPTION	The client supports encryption.
FV_HAS_SOUND	The client has sound.
FV_HAS_DEBUG_OUTPUT	The client can display CALL_DEBUG_OUTPUT messages.
FV_HAS_GZIP	The client can load RTYPE_GZIP resources.

Features with one 8bit argument.

Feature	Description
FV_SCREEN_BPP	The Bits Per Pixel of the screen (Normally 16 or 32).
FV_MOUSE_BUTTONS	Number of mouse buttons (should only be present together with FV_HAS_MOUSE).
FV_KEYBOARD_KEYS	Number of keyboard keys (should only be present together with FV_HAS_PC_KEYBOARD or FV_HAS_VIRTUAL_KEYBOARD).
FV_ALPHA_BLEND	Tells how the client handles alpha blending; 0 For none, 1 = 1bit alpha, 2 = real alpha but 1bit will be faster, 3 = real alpha only.

Features with one 16bit argument.

Feature	Description
FV_SCREEN_HEIGHT	Height of the screen in pixels.
FV_SCREEN_WIDTH	Width of the screen in pixels.
FV_MAX_PACKET_SIZE	Max packet size the client can handle.
FV_IMAGE_SLOTS	Number of images that can be created.
FV_SOUND_SLOTS	Number of sounds that can be created.
FV_MAX_IMAGE_WIDTH	The maximum width and height (in pixels) of a images on the client.

Features with one 32bit argument.

Feature	Description
`FV_DATA_SIZE`	Approx size of Data memory in KBytes. Client guarantees that at least this much memory can be mapped using `CALL_MAP_MEM`. 0 means unspecified.
`FV_PROGRAM_SIZE`	Size of Program memory, in number of instructions.
`FV_RESOURCE_SIZE`	Total size of resources that can be stored in persitant memory, in KBytes. 0 means unspecified.
`FV_MEMORY_SIZE`	Total size of runtime memory available to the swimmer client, in KBytes - this includes data memory and all loaded images and sounds.
`FV_STACK_SIZE`	Maximum size of Execution Unit stack.

Features with a list of (8bit) arguments. First byte after the feature value is number of bytes that follows.

Feature	Description
`FV_SOUND_FORMATS`	A list of sound formats that can be created with `CALL_CREATE_RESOURCE`
`FV_IMAGE_FORMATS`	A list of image formats that can be created with `CALL_CREATE_RESOURCE`
`FV_CUSTOM`	Any feature value that is >= FV_CUSTOM can be used for client specific / application specific features.

6. List of Native Functions

All functions takes arguments in R[0] and forward, but never more than 6 arguments.

Some functions *returns* a value i r0.

Some functions can cause *exceptions*.

Some functions generate a `VEC_RESULT`.

6.1. `CALL_NOOP`

Does nothing. This is initially the target call of `VEC_VBLANK`.

6.2. `CALL_SEND_EVENT`

R[0] = Event number
r1-r3 = Arguments

Sends an event to the server. The event number will be stored in the `PARAM` field in the Packet, and *`R[0]`* will be replaced by the current value of `IO_TIMER`. This call is the target of most event vectors at client startup, so that key presses and mouse clicks are sent to the server if not handled by swimmer code.

6.3. `CALL_SEND_USER_EVENT`

R[0] = Event number
R[1] = Number of arguments
r2-r*n* = Arguments

Sends an custom event to the server. The event number will be stored in the `PARAM` field in the Packet, and R[1] values, starting from the register r2, will be the arguments to the event. Note that R[1] can be zero.

6.4. `CALL_SEND_DATA`

R[0] = Source address
R[1] = Length (in words)

Sends data to the server.

Throws: EXCP_MEM_READ if address is illegal.

6.5. `CALL_MEMCPY`

R[0] = Target address
R[1] = Source address
R[2] = Size (in words)

Standard memcopy in clients (virtual) memory.

Throws: EXCP_MEM_READ or EXCP_MEM_WRITE if addresses are illegal.

6.6. `CALL_MEMSET`

R[0] = Target address

R[1] = Value
R[2] = Size (in words)

Standard memset in clients (virtual) memory.

Throws: EXCP_MEM_WRITE if address is illegal.

6.7. `CALL_QSORT`

R[0] = Address of list to be sorted
R[1] = Number of elements
R[2] = Size of one element (in words)
R[3] = Offset to key element (in words)

Sort a memory area using a certain value as an (integer) key. The offset is calculated from the start of the element.

Throws: EXCP_MEM_READ if address is illegal, EXCP_NATIVE_CALL if Offset is outside element.

6.8. `CALL_ENTER_LINEMODE`

R[0] = Target and source address for entered text
R[1] = Maximum length of text
R[2] = Offset to text to be used as heading, or 0 for default heading
R[3] = The type of text that should be entered

Ask client host for a line of text, in whichever way text input is handled on the device. May temporarily switch out the main screen of the client. Will generate a `VEC_TEXTLINE` event when input is finished.

Type should be one of `INP_TEXT`, `INP_NUMBER`, `INP_LETTERS`, `INP_WORDS`, `INP_PASSWORD`, `INP_PHONE_NUMBER`, `INP_EMAIL`

`Throws:EXCP_MEM_READ if addresses are illegal, EXCP_MEM_WRITE if result overwrites illegal address, EXCP_NATIVE_CALL if Type is unknown.`

6.9. `CALL_CREATE_RESOURCE`

R[0] = Address of 128bit Hash uniquely identifying resource, or 0
R[1] = Address to textual name of resource, or 0
R[2] = Address of resource data to create from, or 0
R[3] = Size of resource data in bytes
R[4] = Type of resource

Creates a named resource in persistent storage. The client should store the resource together with the current *Application ID* so that different applications can use the same resource name without collisions.

The resource is identified by a hash and/or a name. One or both of these must be specified. The hash is normally an MD5 of the resource contents. The name is normally a descriptive name so that the resource can be referred to without knowing the contents.

Pointer to resource data may be zero, in which case the resource is created empty, and is expected to be filled by subsequent data packets to resource memory.

The type identifies the type of resource created, for instance `RTYPE_PNG` or `RTYPE_WAV`.

Throws:EXCP_MEM_READ if addresses are illegal.

6.10. `CALL_LOAD_RESOURCE`

R[0] = Address of 128bit Hash uniquely identifying resource
R[1] = Address to textual name of resource
R[2] = Target address
R[3] = File offset (in bytes)
R[4] = Number of bytes to read (0 means to end of file)

Load a resource into memory. One or both of ID and name must be specified. Resources should preferably only be loaded into 8bit memory, since otherwise the endianness of the host comes into play.

The resource is matched against name first, then hash. If only hash is specified then client may look for resources saved by other applications.

Result: R[2] = RC_SUCCESS or RC_ERR_RES_NOT_FOUND, R[3] = Target address

6.11. `CALL_CLEAR_AREA`

R[0] = X-position
R[1] = Y-position
R[2] = Width
R[3] = Height
R[4] = Color (in RGB format)

Clear a portion of the screen (or current render target) to the specified color. The X and Y position is divided by `IO_XDIVIDER` and `IO_YDIVIDER` respectively. Does not use alpha blending.

6.12. `CALL_CREATE_IMAGE`

R[0] = Image number to (re)define
R[1] = Memory address of pixels (Width * Height pixels)
R[2] = Memory address of Palette (256 * 4 bytes)
R[3] = Width
R[4] = Height
R[5] = Flags

Create an image (that can later be used as a blitting source). A tiling with a single tile that covers the whole image is automatically defined.

A client has a limited number of image slots, which should be reported to the server through the client features packet.

If palette is specified, the pixel data is assumed to be 8bit, otherwise 32bit.

Valid flags are;

`IMG_RENDERTARGET` to create an image that can be used as a render target. This may imply that any alpha channel is ignored.
`IMG_ALPHA` to indicate that an alpha channel should be used.

Throws: EXCP_MEM_READ if addresses are illegal, EXCP_NATIVE_CALL if Image number is not an image slot or if image is too large.

6.13. `CALL_CREATE_IMAGE_FR`

R[0] = Image number to (re)define
R[1] = Address of 128bit Hash uniquely identifying resource
R[2] = Address to textual name of resource
R[3] = Flags

Similar to `CALL_CREATE_IMAGE` except a resource in a known format (normally PNG) is used as the source. A tiling with a single tile that covers the whole image is automatically defined.

This type of image can not be a render target, and the alpha channel is deducted from the file itself, so the `IMG_RENDERTARGET` and `IMG_ALPHA` flags are ignored. The only valid flag is `IMG_LAZY`, which indicates that the client does not have to create the image unless it is actually rendered.

The resource is matched against name first, then hash. If only hash is specified then client may look for resources saved by other applications.

RESULT: `R[2] = RC_SUCCESS` if resource was found or `RC_ERR_RES_NOT_FOUND` if not , R[3] = Image number

6.14. `CALL_SET_RENDER_TARGET`

R[0] = Image number to use as target, or -1 to use the screen (the default)

This lets swimmer use a specific image as render target instead of the screen. The image must have been created with the `IMG_RENDERTARGET` flag.

6.15. `CALL_DEFINE_IMAGE_TILING`

R[0] = Image number to define tiling for
R[1] = Width of a single tile, or list of widths for a variable width tileset
R[2] = Height of a single tile
R[3] = Number of tiles (laid out left to right, then top to bottom)
R[4] = Flags

Define a tiling for an image so that parts of it can be drawn to the screen. A tiling can be used to define the layout of fonts, sprites, background graphics etc (so *tile* may not be the best name). Tiles are laid out left to right then top to bottom. Possible flags are;

TS_WIDTHPTR - Means R[1] points to list of widths

Throws: EXCP_MEM_READ if TS_WIDTHPTR is set and address is illegal, EXCP_NATIVE_CALL if Image number is invalid or does not represent an image, if Width or Height doesn't fit image.

6.16. `CALL_RENDER_IMAGE`

R[0] = The source image to render from
R[1] = Which tile to render
R[2] = Target X-position
R[3] = Target Y-position

Render a single tile from an image to the screen (or the current render target). The X and Y position is divided by `IO_XDIVIDER` and `IO_YDIVIDER` respectively.

Returns: The width of the tile.

6.17. CALL_SET_ENCRYPTION

R[0] = KeyId, or -1 to use the any available key
R[1] = Seed value
R[2] = Encryption type

Start or stop an encryption session with the server.

If `-1` is specified as key, any available (but normally the first) key is used.

Note that the encryption is not actually turned on until after the RESULT was generated, otherwise the server can not see which KeyId is used for encryption, and thus not find the correct key to actually decrypt the packets coming from the client.

The seed value is combined with the key to form a session key.

The type decides which type of encryption should be active from now on:

`ENCR_BLOWFISH_FROM_RES` - Start blowfish encryption using a numbered key resource as key.
`ENCR_BLOWFISH_FROM_MEMORY` - Start blowfish encryption using a key in memory.The KeyId argument is the address of the key.
`ENCR_RSA_FROM_RES` - Start RSA encryption using a numbered key resource as key.
`ENCR_NONE` - End any current encryption sessions and go back to unencrypted traffic.

When you turn off encryption using `ENCR_NONE`, all other arguments are ignored.

RESULT: R[2] = `RC_SUCCESS` or `RC_ERROR` if key was not found, R[3] = The KeyId if found.

6.18. CALL_LOAD_PACKAGE

R[0] = Address of 128bit Hash uniquely identifying package

Try to load an `RTYPE_PACKAGE` resource, and apply all packets contained in it as though they came from the server. Note that there are no corresponding SAVE_PACKAGE, packages are created as normal resources (of type `RTYPE_PACKAGE`).

6.19. CALL_MAP_MEM

R[0] = Bank
R[1] = Minimum size (in words)
R[2] = Type and minimum bitsize.

Create a new memory area with the given size and format, and map it to the given bank. If the bank is already mapped to a memory area, it is discarded.

6.20. CALL_DEBUG_PRINT

R[0] = Address of text string

Print a line of text to the debug console if available, using a printf-like format.

6.21. CALL_ALERT

R[0] = Flags
R[1] = Message

Alert the user through a combination of sound, vibration and light. This will be different for different host hardware so the flags are only a recommendation.

Flags can be a combination of these;

`ALERT_SOUND` = Some sort of beep or noise
`ALERT_VIBRATE` = Vibration
`ALERT_BACKLIGHT` = Turn on screen backlight
`ALERT_LED` = Blink Indicator light
`ALERT_POPUP` = Unminimize and unhide if client is hidden.

`ALERT_DISCRETE` = Short, one-shot
`ALERT_NORMAL` = Normal length (two double beeps or similar)
`ALERT_NOISY` = Long and annoying (but no more than 10 sec)
`ALERT_PROGRESSIVE` = Go from discrete to noisy (but no longer than 10 sec)

The message should only be displayed on clients that have a special area for it, that is not part of the client screen. The normal cases are popup bubbles for PC clients and a LED-display on special devices like phones or LED signs.

6.22. CALL_CREATE_SOUND

R[0] = Sound number
R[1] = Address of sound data
R[3] = Length of sound data
R[4] = Frequency in Hz
R[5] = Format

Create a sound from sample data in memory. Format can currently only be;

`SFMT_PCM16S` = 16bit, signed, single channel PCM sample data

6.23. CALL_CREATE_SOUND_FR

R[0] = Sound number
R[1] = Address of 128bit Hash uniquely identifying resource
R[2] = Address to textual name of resource

Creates a sound from a resource file.

The resource is matched against name first, then hash. If only hash is specified then client may look for resources saved by other applications.

6.24. CALL_PLAY_SOUND

R[0] = Sound number
R[1] = Channel, or -1 for any free channel
R[2] = Position, or 0 to play right away

Play a previously created sound. The position argument is only supported on some hardware.

6.25. CALL_REMOVE_RESOURCES

R[0] = Address of 128bit Hash uniquely identifying resource

R[1] = Address to textual name of resource
R[2] = Flags

This will remove all resources matching either or both of Hash and Name. If both criteria is zero, all resources will be removed. If both criteria are specified, resources matching name will be removed, *except* the one that also matches hash (used to delete all old resources by name, but saving the current one).

Flags can be;

`RR_UNUSED` - Remove only unused resources, meaning resources that has not been accessed since the last RESET of the client.

6.26. `CALL_GET_FEATURE`

R[0] = Feature Index

Returns the value of a specific client feature. Used for instance by an offline application to learn the width and height of the screen.

6.27. `CALL_RESET`

R[0] = Flags

Perform a client reset according to the given flags.

`RES_SEND_FEATURES` = Send the feature packet to the server again
`RES_CLEAR_SCREEN` = Clear the screen to black
`RES_DELETE_MEMORY` = Remove and free all data memory mappings
`RES_CLEAR_LOADED_RESOURCES` = Unload all created images and sounds
`RES_CLEAR_PROGRAM` = Clear program memory
`RES_CLEAR_VECTORS` = Clear vector memory
`RES_CLEAR_REGISTERS` = Clear all registers

These flags are guranteed to be represented by bits above bit 15, and so is not used when the flag argument is 0xFFFF;

`RES_QUIT` = Exit the swimmer client
`RES_CLEAR_SRAM` = Clear SRAM

6.28. `CALL_GENERATE_RANDOM`

R[0] = Target address.
R[1] = Length in words of target memory.

Generates true random values (for use with encryption).

RESULT: R[2] = RC_SUCCESS or RC_ERROR if there was not enough entropy to create the required number of random values.

6.29. `CALL_SET_MULTICALL`

R[0..5] = Argument flags and increment
R[6] = Target Call

Sets up a multicall. TargetCall is the native call that should be called in succession by `CALL_DO_MULTICALL`. The other arguments contains information on how each argument to the target call should be handled; the top 8 bits specify flags and the bottom 24 bits is a signed integer that should be added to that argument for each successive call.

Flags:

`MC_INDIRECT` = This argument is a pointer into data memory
`MC_XDIV` = This argument should be divided by the current `IO_XDIVIDER` before each use
`MC_YDIV` = This argument should be divided by the current `IO_YDIVIDER` before each use
Note the use of *current* above; the dividers will be read and saved when CALL_SET_MULITICALL is called, so you can change the dividers before the actual call.

6.30. `CALL_DO_MULTICALL`

R[0..5] = Initial arguments
R[6] = Number of time to call the TargetCall

Call the previously set up TargetCall a number of times in succession. After each call the arguments will be modified according to the previous `CALL_SET_MULTICALL`.

7. Debugging (obsolete)

7.1. Setup

- Server sends a `DebugPacket` to a client (this step is optional since client can initiate debugging itself, normally on startup). The *id* field may be set to any value at this point.
- The client opens a new socket connection to the server and sends *two* debug packets, one to the present connection and one to the new connection. The *id* field needs to be set to the same value - either a completely random value or a client specific ID of some kind.
- When the server receives the `DebugPacket` on the new connection it uses the *id* to find out which client is on the other end. This new connection is the one used for all debug commands.
- To start a debug session, a debugger connects to server in the same way as a client, but instead of sending a `FeaturePacket`, it sends a DebugPacket with *id* set to `0xFFFFFFFF.`
- A debugger can only send server commands until it has connected to a specific client using the *debug* command. At that point, all client commands are redirected to the selected client.
- The server listens to all connected debuggers, but only one debugger may be connected to a specific client at the same time.

7.2. Commands to server

These commands are interpreted directly by the server and requires no communication with a client. Some of the commands require you to be connected to a client just because they need an active client.

`debug` {*`id`*}

Connect to the given client.

`list`

List all connected clients and their IDs

`pc` {*`address`*}

Display file and line associated with a specific location in program memory for the current client.

`src` {*`sourcefile`*} {*`line`*}

Display program memory offset associated with a specific assembler file and line for the current client.

`symbols`

Show all symbols defined for the current client.

`labels`

Show all labels defined for the current client.

7.3. Commands to client

Commands that a swimmer client listens to.

`c`

Continue execution after a break or exception

`br` {*`address`*}

Set a breakpoint on a program memory address

`bd` {*`address`*}

Deletes a breakpoint from a program memory address

`bl`

List all breakpoints

`bc`

Clears all breakpoints

`wr` {*`address`*}

Set a read watch on a data memory address

`ww` {*`address`*}

Set a write watch on a data memory address

`wd` {*`address`*}

Deletes a watch from a data memory address

`wl`

List all watchpoints

`wc`

Clears all watchpoints

`x`

Show the values of all registers

`md` {*`adress0`*} {*`adress1`*}

Show contents of data memory between two addresses

`mp` {*`adress0`*} {*`adress1`*}

Show contents of program memory between two addresses

`mv` {*`adress0`*} {*`adress1`*}

Show contents of vector memory between two addresses

`pd` {*`address`*} {*`value`*}

Put a value into data memory

`mp` {*`address`*} {*`value`*}

Put a value into program memory

`pv` {*`address`*} {*`value`*}

Put a value into vector memory

`sl`

Step to the next program memory address

`si`

Step one instruction, following branches

7.4. Results from client

The results from clients are very similar to the commands it gets sent.

`c`

Response to 'c' command

`br` {*`address`*}

Indicates a breakpoint location. Response to 'br' and 'bl' commands

`bd` {*`address`*}

Response to 'bd' command

`bc`

Response to 'bc' command

`wr` {*`address`*}

Indicates a read watchpoint location. Response to 'wr' and 'wl' commands.

`ww` {*`address`*}

Indicates a write watchpoint location. Response to 'ww' and 'wl' commands.

`wd` {*`address`*}

Response to 'wd' command.

`wc`

Response to 'wc' command

`x` {*`register`*} {*`value`*}

Show the value of a register. Response to 'x', 'si' and 'sl' commands.

`md` {*`adress0`*} {*`adress1`*} {*`value`*...}

Shows contents of data memory between two addresses. Response to 'md' and 'pd' commands.

`mp` {*`adress0`*} {*`adress1`*} {*`value`*...}

Shows contents of program memory between two addresses. Response to 'mp' and 'pp' commands.

`mv` {*`adress0`*} {*`adress1`*} {*`value`*...}

Shows contents of vector memory between two addresses. Response to 'mv' and 'pv' commands.

Appendix A. Assembler Examples

1. Drawline

```
    ; Arguments
    @defreg "rX0", 0
    @defreg "rY0", 1
    @defreg "rX1", 2
    @defreg "rY1", 3
    @defreg "rCOLOR", 4
draw_line:

    @defreg "rDX", 5
    @defreg "rDY", 6

    sub     rDX,rX1,rX0
    sub     rDY,rY1,rY0

    ; Return if line has zero length
    bne     rDX,.notzero
    bne     rDY,.notzero
    ret
.notzero

    bge     rDX,@+1
    neg     rDX
    bge     rDY,@+1
    neg     rDY

    blt     rDY,rDX,.xline

    ; y direction is largest

    blt     rY0,rY1,.ok

    ; Swap x0,y0 with x1,y1
    swap    rY0,rY1
    swap    rX0,rX1
.ok

    sub     rDX,rX1,rX0
    asl     rDX,8
    sub     rDY,rY1,rY0
    div     rDX,rDY             ; deltax = (x1-x0)*256/(y1-y0)
    asl     rX0,8               ; x0 = x0 * 256

    mul     rY0,fbWidth
    add     rY0,frameBuffer

    @defreg "rPTR", 2

.yloop
    asr     rPTR,rX0,8
    add     rPTR,rY0
```

```
        store   [rPTR],rCOLOR

        add     rX0,rDX
        add     rY0,fbWidth

        dbgt    rDY,1,.yloop

        ret

    .xline
        ; x direction is largest

        blt     rX0,rX1,.ok2
        ; Swap x0,y0 with x1,y1
        swap    rY0,rY1
        swap    rX0,rX1
    .ok2

        sub     rDY,rY1,rY0
        asl     rDY,8
        sub     rDX,rX1,rX0
        div     rDY,rDX              ; deltay = (y1-y0)*256/(x1-x0)
        asl     rY0,8                ; y0 = y0 * 256

        add     rX0,frameBuffer

        @defreg "rPTR", 2
    .xloop
        asr     rPTR,rY0,8
        mul     rPTR,fbWidth
        add     rPTR,rX0

        store   [rPTR],rCOLOR

        add     rY0,rDY
        add     rX0,1

        dbgt    rDX,1,.xloop

        ret
```