KIPPER API Technical Reference

Document History

Date	Author	Changes
2009-09-22	Jonno Downes	Initial baseline
2009-10-04	Jonno Downes	Removed VBI
2009-10-28	Jonno Downes	Added web application server functions

Contents

Introduction

The KIPPER application programming interface (or API) is a set of functions that allow C64 programs to communicate over an IP network without being tied to a specific hardware device.

The KIPPER API is intended to

- Be simple for developers to use, regardless of what their preferred development tools.
- Allow programs loaded from disk (or via tftp) to use code stored in cartridge ROM without being tied to any specific ROM image.
- Remove requirement for each program to independently configure MAC and IP addresses.
- Provide a hardware abstraction layer to allow independent development of network programs and network interface devices - KIPPER programs should work as easily with a (as yet undeveloped) wifi cartridge as with the current cs8900a based RR-NET compatible devices (as long as each cartridge implements the appropriate KIPPER functions).

The initial implementation of the KIPPER API is part of the “kipper” project, which is based on the “ip65” library, and includes a ruby based tftp server with some non-standard extensions. However it is important to keep a distinction between the definition of the KIPPER API, and the implementation of that API within the kipper project – other cartridge developers are free to implement the KIPPER in their own cartridges, using whatever underlying library they choose, and as long as they implement each function defined by the KIPPER API, any program coded to that API should work.

In the remainder of this document, text *in italics* contains information specific to the kipper implementation of the KIPPER API, where other implementers are free to do things differently. All other text refers to KIPPER API as it should be in every implementation.

Using the API

Detecting and activating the KIPPER API

If the KIPPER API is installed and active (banked in), the string “KIPPER” (hex $4B $49 $50 $50 $45 $52) can be read from location $8009..$800e.

IP stack initialisation

Once the KIPPER API has been located, call KPR_INITIALIZE. This function takes no inputs, and the only result returned is the carry flag is set on error, and clear otherwise.

The IP initialisation process will do the following

- Configure MAC address, IP address, netmask , default gateway and DNS server (*the kipper cartridge does this via DHCP)*
- Any other internal housekeeping *(the kipper cartridge sets up to use the timers on CIA #2 ($DD0x). Timers A & B are a combined 16-bit count-down of milliseconds.)*

For the kipper cartridge, the most likely causes of failure are:

- *No ethernet controller being found, in which case the next call to KPR_GET_LAST_ERROR will return $85 - KPR_ERROR_DEVICE_FAILURE*
- *No DHCP server responds during DHCP initialisation, in which case the next call to KPR_GET_LAST_ERROR will return $81 - KPR_ERROR_TIMEOUT_ON_RECEIVE*

Periodic Processing

In order to detect and respond to inbound IP packets, the KIPPER Periodic Processing Vector ($8012) should be called regularly – at least a few times each second. The amount of time each call to this vector will vary, depending on whether or not an inbound message is waiting.

API Conventions

All KIPPER functions are called by a JSR to $800F with the Y register loaded with a function number.

Where a function has 1 input, it will be passed in via the A & X registers. Where a function has 1 output, that output will be passed via the A & X registers.

Where a function has more than 1 input, or returns more than 1 output , AX should be set with the address of a buffer that can be used for passing multiple parameters. Addresses are passed in with A=low byte, X=high byte (e.g.$1234 would be passed in as A=$34, X=$12.) The format of this buffer will vary for each function, although no function requires a parameter buffer of more than $20 bytes so a single area of that size can be reserved for this purpose.

Errors

All KIPPER functions set the carry flag if there is an error, and clear it if there was no error. If the carry flag is set, the KPR_GET_LAST_ERROR function can be called to retrieve a 1 byte error code indicating what went wrong (NB – the value returned by KPR_GET_LAST_ERROR is not cleared by successful function calls, it always carries the code indicating the last failure).

KIPPER Functions

API housekeeping functions

Number	Description	Parameters *
$01	**KPR_INITIALIZE** Should be called once by each program, prior to using any other KIPPER functions *In the kipper implementation, this sets up internal structures, and also injects an IRQ handler into $314* *The kipper implementation checks whether or not the RUN/STOP key has been pressed, and if it has will abort the DHCP configuration (and the next call to KPR_GET_LAST_ERROR will return $86 – Aborted by user.*	**Inputs**: none **Outputs**: none
$02	**KPR_GET_IP_CONFIG** Returns a pointer to a table containing the current IP configuration. The data in this table should not be modified.	**Inputs**: n/a **Outputs**: AX contains pointer to an IP Configuration Structure
$0F	**KPR_DEACTIVATE** This routine should be called if an program has finished using the KIPPER API and wants to reclaim RAM for other purposes. *On kipper cartridges, this function restores the previous value of the IRQ vector $314*	**Inputs**: none **Outputs**: none
$FF	**KPR_GET_LAST_ERROR** Returns an error code that specifies the reason for the last failure by any KIPPER function (which may not have been the last KIPPER function called)	**Inputs**: none **Outputs**: A = error code (per table below)

Transport layer functions

$10	**KPR_UDP_ADD_LISTENER** This function takes a port number and a callback address – whenever a UDP packet arrives on the specified port, then the specified callback routine will be executed.	**Inputs**: AX contains pointer to a UDP Listener Structure **Outputs**: None
$11	**KPR_GET_INPUT_PACKET_INFO** This routine returns information about the last IP packet to arrive. If it is called within a TCP or UDP Listener callback routine, then the packet being described is the one which triggered the callback. The structure returned by this function is in the same format as the structure required as input to KPR_SEND_UDP_PACKET, this makes it easy to create callback routines that generate replies.	**Inputs**: AX contains pointer to a buffer where the UDP /TCP Packet Structure can be written **Outputs**: specified buffer has Packet Structure written to it.
$12	**KPR_SEND_UDP_PACKET** Send a UDP packet to a remote host. *On kipper cartridges, this function requires there already be an entry in the ip65 ARP table with the MAC address corresponding to the specified IP. If there is no such ARP entry, then the call to KPR_SEND_UDP_PACKET will fail, but an ARP request will be sent out, so future attempts to communicate with the requested IP will succeed. Note: in order for replies to the ARP request to be seen, and the ARP table updated, programs must call the KIPPER Periodic Processing Vector ($8012) at the original attempt to send the UDP packet which failed but triggered the ARP request being sent, and the next attempt to send the same UDP packet.* *Note that even if the call returns successfully, there is no guarantee that the packet has been transmitted intact across the network and arrived at the destination, hence UDP programs generally implement acknowledgment, timeout and retransmission mechanisms – if this is in place then the special case outlined above (where ARP resolution is required before the packet is transmitted) will be covered as well and no additional handling is*	**Inputs**: AX contains pointer to a UDP Packet Structure **Outputs**: none

	required.	
$13	**KPR_UDP_REMOVE_LISTENER** This function takes a port number – this UDP port will no longer be listened on.	**Inputs**: AX contains number of port that will no longer be listened on. **Outputs**: None
$14	**KPR_TCP_CONNECT** This function will takes an IP address, a port number and a pointer to a callback routine. If the IP address passed in is "0.0.0.0", this is treated as a request to act as a server, and the specified port will be listened on. The call will not return until either an inbound client connects, OR an error occurs (including the user aborting the listen by keypress). If any other address is specified, this is treated as a request to act as a client, and a TCP connection will be attempted to the specified IP address and port number - a unique port number will be used for the local side of the connection. Whether a remote IP is passed in (client mode) or not (server mode), whenever any data (excluding any empty 'ACK only', or out of sequence, packets) arrives from the remote end, the routine specified by the 'callback' pointer will be executed. If the connection is terminated by the other end, a callback will be generated with a payload length of $ffff.	**Inputs:** AX contains pointer to TCP Connection Structure. **Outputs**: none
$15	**KPR_SEND_TCP_PACKET** Sends data via to specified TCP connection. The connection must have already been set up (via KPR_TCP_CONNECT). Data is sent immediately and must fit into a single datagram i.e. (there is no buffering or splitting of input into multiple datagrams).	**Inputs**: AX contains pointer to TCP Send Structure. **Outputs**: None
$16	**KPR_TCP_CLOSE_CONNECTION** Closes the current TCP connection.	**Inputs**: None **Outputs**: None

TFTP Functions

$20	**KPR_TFTP_SET_SERVER** Sets the IP address of the TFTP server that all subsequent TFTP transfers will occur with.	**Inputs**: AX contains pointer to a TFTP Transfer Server Structure **Outputs**: none
$22	**KPR_TFTP_DOWNLOAD** Download the specified filename from a tftp server. This uses the standard tftp download opcode and hence will work with any tftp server. *There is no bounds checking on this function – it is up to the caller to ensure that the file will fit into the specified buffer. If the file is too large it is likely to overwrite kipper code or system variables with unpredictable results.*	**Inputs**: AX contains pointer to a TFTP Transfer Parameter Structure **Outputs**: The specified file will be downloaded into the buffer pointed at by KPR_TFTP_POINTER . If the address passed in was $000 then the first 2 bytes of the file are used to determine the load address and KPR_TFTP_POINTER will be updated with that address
$23	**KPR_TFTP_ CALLBACK_DOWNLOAD** Download the specified filename from a tftp server. This uses the standard tftp download opcode and hence will work with any tftp server. This function will generate a callback when each block arrives from the server. This can be used to (for example) write files of that are too big to fit into RAM to disk. All blocks except the last block will be 512 bytes long. The last block will be less than 512 bytes long (and will be 0 bytes long if the length of the file being downloaded is a multiple of 512 bytes). So the way the callback routine should test whether the current block is the last one in the transfer is to see test byte 1 in the input buffer – if it is a $02 then there are more blocks to come, if it is a $00 or $01 then this is the last block.	**Inputs**: AX contains pointer to a TFTP Transfer Parameter Structure **Outputs**: The specified file will be downloaded in 512 byte blocks. When each block arrives, the routine specified by KPR_TFTP_POINTER will be called with AX set to point at a buffer containing: Bytes 0/1 = length of block Bytes 2..514 = block data.

$24	**KPR_TFTP_ UPLOAD** Send a file with the specified filename to a tftp server. This uses the standard tftp download opcode and hence will work with any tftp server. This function will send data from the address specified in the KPR_TFTP_POINTER parameter. The total number of bytes to send must be specified in the KPR_TFTP_FILESIZE parameter.	**Inputs**: AX contains pointer to a TFTP Transfer Parameter Structure **Outputs**: The specified file will be sent to the specified tftp server.
$25	**KPR_TFTP_ CALLBACK_UPLOAD** Send a file with the specified filename to a tftp server. This uses the standard tftp download opcode and hence will work with any tftp server. This function will call the user provided function once for each 512 block that needs to be sent to the server. Note that the filename passed in is only used to inform the tftp server what name to save the uploaded data as. This function will not open a file from a local disk and send it – it is up to the calling program to provide the function of reading from the disk. The callback routine needs to be implemented as follows: 1) When it is called, AX will be pointing at a 512 byte buffer that the next block of data is to be written to. 2) The routine must copy up to 512 bytes of data into that buffer, and then set AX to be the number of bytes actually copied (i.e. should be between 0 and 512). 3) The TFTP protocols signals the "end of file" by sending a block with less than 512 bytes. Therefore the last block sent must be less than 512 bytes. If the file being sent is a multiple of 512 bytes, then a final block of 0 bytes must be sent to transmission has finished.	**Inputs**: AX contains pointer to a TFTP Transfer Parameter Structure **Outputs**: The specified file will be sent in 512 byte blocks. The routine specified by KPR_TFTP_POINTER will be called once for each block that needs to be sent, with AX pointing at a buffer that needs to be filled with the next block of data to be sent.

Other network functions

$30	**KPR_DNS_RESOLVE** Resolve a string containing a hostname OR an IP address in "dotted quad" format (e.g. "192.168.1.1") into a 32 bit IP address. This requires a DNS server that supports recursive queries (which almost all DNS servers will) *The kipper implementation checks whether or not the RUN/STOP key has been pressed, and if it has will abort the DNS resolution (and the next call to KPR_GET_LAST_ERROR will return $86 – Aborted by user.*	**Inputs**: AX contains pointer to a DNS Parameter Structure **Outputs**: First 4 bytes of the DNS Parameter structure updated to contain the IP address.
$31	**KPR_DOWNLOAD_RESOURCE** Downloads (via http or gopher) a "resource", e.g. a file. Specified URL must be a valid http:// or gopher:// URL in ASCII. (e.g. http://www.example.com:8080/foo.xml) Any 'control character' (i.e. <$20, including CR ($0D), LF($0A) or NUL ($00) will be treated as the end of the URL. This implementation has the following limitations: • http and gopher only (not ftp or https) • authentication is not supported (e.g. http://user:pass@example.com/protected/ will NOT work) • no entity encoding/decoding • The result of a HTTP download will include the full HTTP response header, i.e. client code will need to interpret status code, follow redirects, skip to "\n\n" to get the actual file contents etc. HTTP downloads are 1.0 compliant (including sending a valid Host: header) The downloaded file will always have a trailing null byte ($00) appended.	**Inputs**: AX contains pointer to a URL Download Structure **Outputs**: The specified resource will be downloaded into the buffer pointed at by KPR_URL_DOWNLOAD_BUFFER (truncated to buffer size)

$32	**KPR_PING_HOST** Sends a "ping" (ICMP echo request) message to a host and reports on how long it took to receive a response. NB – the response time is measured by the TCP stack timer, which is neither very accurate nor very granular.	**Inputs**: AX contains pointer to IP address of host to ping **Outputs**: AX will contain the time (in milliseconds) between pinging the host and receiving a response.

File Access functions

$40	**KPR_FILE_LOAD** Load the specified filename from disk. KPR_FILE_ACCESS_DEVICE should be set as follows: $00 = whatever device was last accessed (or the 'default' drive if this is the first access) $01 = first drive on system (i.e. drive #8 on a C64) $02 = second drive on system (i.e. drive #9 on a C64) Etc. *There is no bounds checking on this function – it is up to the caller to ensure that the file will fit into the specified buffer. If the file is too large it is likely to overwrite kipper code or system variables with unpredictable results.*	**Inputs**: AX contains pointer to a Disk Access Parameter Structure **Outputs**: The specified file will be loaded into the buffer pointed at by KPR_FILE_ACESS_POINTER . If the address passed in was $000 then the first 2 bytes of the file are used to determine the load address and KPR_FILE_ACESS_POINTER will be updated with that address. The size of the loaded file will be saved in KPR_FILE_ACCESS_FILESIZE

Web Application Server functions

$50	**KPR_HTTPD_START** Start the web application server (aka HTTP daemon). This function will return ONLY after the web application server stops, i.e. either due to an error or after the runstop/restore key being pressed.	**Inputs**: AX contains pointer to the httpd callback routine (executed for each inbound http request). **Outputs**: Since this function only returns after the web server stops, the carry flag will be set and the reason for exit can be retrived by calling KPR_GET_LAST_ERROR

<table>
<tr>
<td>$52</td>
<td>
KPR_HTTPD_GET_VAR_VALUE
To be used by httpd callback routines to check value of query string variables.

Current implementation has the following limitations:

Only the first letter in each variable name is significant, i.e. ‘e’ and ‘example’ are treated as a single variable (although case is significant – ‘e’ and ‘E’ are different variables)
Only variables in the query string can be retrieived, i.e. if you have a html form, you should use method=GET not method=POST

The following ‘special’ variables can be retrieved:
$01 = ‘method’ (e.g. “GET” or “POST”)
$02 = path (e.g “/foo.html”)

For example, if a client made a HTTP request of:

<pre>GET /example.html?foo=bar HTTP/1.0
User-Agent: IP65 v0.9.1
Host: c64.example.com</pre>
Then the following values will be returned

<table>
<tr><td>Call with A =</td><td>Returns</td></tr>
<tr><td>$01</td><td>AX points to “GET”,$00</td></tr>
<tr><td>$02</td><td>AX points to “/example.html”,$00</td></tr>
<tr><td>‘f’</td><td>AX points to “bar”,$00</td></tr>
<tr><td>‘F’</td><td>Error : Carry flag set</td></tr>
</table>
</td>
<td>
Inputs: A contains first char of variable name
Outputs:
AX points at null terminated string containing variable value.
</td>
</tr>
</table>

Printing Functions

All the KPR_PRINT_* functions use the CHROUT kernal routine for output and do not modify the output device number, so programs can if they choose, change the output to go somewhere other than the screen via calling the kernal CHKOUT first.

$80	**KPR_PRINT_ASCIIZ** This routine prints the specified string to the screen	**Inputs**: AX contains pointer to null terminated string to be printed to screen **Outputs**: none
$81	**KPR_PRINT_HEX** This routine prints a single byte as two hex digits.	**Inputs**: A = byte digit to be displayed on screen as (zero padded) hex digit **Outputs**: none
$82	**KPR_PRINT_DOTTED_QUAD** This routine prints a 32 byte IP address in "dotted quad" format. For example, $C0A80102 will be displayed as "192.168.1.2"	**Inputs**: AX contains pointer to 32 bit IP address **Outputs**: none
$83	**KPR_PRINT_IP_CONFIG** Prints a summary of the currently active IP configuration. The exact format of this output can vary by implementation.	**Inputs**: none **Outputs**: none
$84	**KPR_PRINT_INTEGER** Prints a 16 bit number as an unsigned (and unpadded) integer.	**Inputs**: AX = 16 bit number to be printed **Outputs**: none

Input Functions

$90	**KPR_INPUT_STRING** This routine returns a user-entered string. The 'periodic processing' routine will be regularly polled while waiting for user input.	$90
$91	**KPR_INPUT_HOSTNAME** This routine returns a user-entered string that contains a hostname. Only characters that are valid in a DNS hostname can be entered (including dots and numbers, but no white space). The 'periodic processing' routine will be regularly polled while waiting for user input.	**Inputs**: none **Outputs**: AX points to a null terminated string containing hostname entered by user. The carry flag is set if nothing was entered.
$91	**KPR_INPUT_PORT_NUMBER** This routine waits for the user to input a number, and returns the value entered as a 16 bit number in AX. The 'periodic processing' routine will be regularly polled while waiting for user input.	**Inputs**: none **Outputs**: AX contains the value entered by the user. The carry flag is set if nothing was entered.

Utility Functions

$A0	**KPR_BLOCK_COPY** This routine copies a block of bytes from one location in memory to another. The copy is done from the bottom up.	**Inputs**: AX contains pointer to a Block Copy Structure **Outputs:** none
$A1	**KPR_PARSER_INIT** This routines sets up a string to be searched for substrings (i.e. by subsequent calls to KPR_PARSER_SKIP_NEXT).	**Inputs**: AX points to a null terminated string. **Outputs:** none
$A2	**KPR_PARSER_SKIP_NEXT** This routine scans through the string previously loaded into the parser (i.e. by calling KPR_PARSER_INIT) and returns once the specified substring has been found. A pointer is updated so successive calls to this routine will return different parts of the string being parsed. If there are no more occurrences of the specified substring, the carry flag will be set (and AX will be whatever the pointer was at *start* of the call, i.e. whatever was the last substring successfully searched for, OR the start of the string, if no substring searches have yet matched).	**Inputs**: AX points to a null terminated string. **Outputs:** AX points to the first byte past the next occurrence of the specified substring within the string that the parser was last initialised with.

* In addition to the outputs specified in this column, each function uses the carry flag to indicate success (clear) or failure (set). If an error occurs (and carry flag is set) then the other outputs are undefined. Any register not mentioned in the outputs should be treated as ‘undefined’ (i.e. there is no guarantee they won’t be modified by the function).

KIPPER Web Applications

Starting the Web Application Server

The KIPPER Web Application Server is started with the KPR_HTTPD_START function. This function does not return while the server is running. Before calling this function, load AX with the address of a routine to be called when each HTTP request is made.

The request callback handler

The request callback routine will be called by the web application server whenever a http request has arrived. This routine must be written to provide the following interface:

Inputs
When called, the none of the flags or registers have any special meanings.
The HTTP request triggering this callback will have been parsed, and query string variables and other components of the request can be accessed by calls to the KPR_HTTPD_GET_VAR_VALUE function.

Outputs
When execution of the callback handler completes (i.e. via an RTS) the registers must be set as follows:
Carry Flag – should be clear
AX – points to a buffer containing the data to be sent back as a response.
Y – indicates the 'type' of the response (which is used to create the HTTP header at that start of the response). Valid values are:

Y	**Response**
$00	No header is created (assumes there is a header in the buffer pointed at by AX)
$01	Normal text – response created with status code 200 (OK), with Content-Type: 'text/text'
$02	Normal html – response created with status code 200 (OK), with Content-Type: 'text/html
$03	Binary file – response created with status code 200 (OK), with Content-Type: 'application/octet-stream'
$04	Response created with status code 404 (Not Found)
$05	Response created with status code 500 (System Error)

HTML templates

The buffer pointed at by AX when the callback routine completes is treated as a null terminated string containing a HTML template (this implies the current version of the API can't be used to send arbitrary binary data).
The following codes have special meaning in the HTML template

Code	Meaning
%$<b>	**Variable Value** : The byte following %$ is treated as the name of variable. The value of the variable of that name is inserted into the output in place of the %$ code.
%:<bbbb>	**Call Routine** : The 4 bytes following %: are treated as being hex digits specifying the address of a routine to call.
%;<bb>	**Call Routine** : The 2 bytes following %; are treated as being hex digits specifying the address of a routine to call.
%?<b>	**If Variable Defined** : The byte following %? is treated as the name of variable. If that variable is defined in the input request query string (even if it is set to a null value) then output continues as normal. If the variable is NOT defined, then output is suppressed until the %. Code appears in the HTML template.
%!<b>	**If Variable Not Defined** : The byte following %! is treated as the name of variable. If that variable is NOT defined in the input request query string (even if it is set to a null value) then output continues as normal. If the variable IS defined, then output is suppressed until the %. Code appears in the HTML template.
%.	**End of Condition** – this code marks the end of a %? or %! condition. Output resumes if it had previously been suppressed (because of a %? which specified an undefined variable, or a %! which specified a defined variable)

Example Web Application

This code implements a simple html form which prompts for a handle (nickname) and a message – if the form is submitted, the values that were input are echoed back.

```
lda <#httpd_callback
ldx >#httpd_callback
ldy #KPR_HTTPD_START
jsr KPR_DISPATCH_VECTOR       ;should never exit
rts

httpd_callback:
  lda <#html
  ldx >#html
  ldy #2 ;text/html
  clc
  rts

html:
```

```
  .byte "<h1>hello world</h1>%?mMessage recorded as
'%$h:%$m'%.<form>Your Handle:<input name=h type=text
length=20 value='%$h'><br>Your Message: <input type=text
lengh=60 name='m'><br><input type=submit></form><br>",0
```

KIPPER Structures

IP Configuration Structure

Used By: KPR_GET_IP_CONFIG

Offset	Size (bytes)	Contents
$00	$06	MAC Address
$06	$04	Local IP address (will be overwritten by DHCP)
$0A	$04	Local netmask (will be overwritten by DHCP)
$0E	$04	Local gateway (will be overwritten by DHCP)
$12	$04	IP address of DNS server (will be overwritten by DHCP)
$16	$04	IP address of DHCP server (will only be set by DHCP initialisation)
$1A	$02	Pointer to ASCIIZ string containing name of device type (e.g. “RR-NET”)

TFTP Server Parameter Structure

Used By: KPR_TFTP_SET_SERVER

Offset	Size (bytes)	Contents
$00	$04	IP address of TFTP server (use $FFFFFFFF to do a broadcast on local LAN)

TFTP Transfer Parameter Structure

Used By: KPR_TFTP_DOWNLOAD, KPR_TFTP_CALLBACK_DOWNLOAD, KPR_TFTP_UPLOAD, KPR_TFTP_CALLBACK_UPLOAD

Offset	Size (bytes)	Contents
$00	$02	Pointer to ASCIIZ filename
$02	$02	For KPR_TFTP_DOWNLOAD this field is a pointer to memory location data to be stored in (set this to $0000 when downloading a file where the first 2 bytes are the memory location the file should be stored in, e.g. a C64 PRG file) For KPR_TFTP_CALLBACK_DOWNLOAD, this field is the address of the routine to be called when each 512 block arrives.
$04	$02	Size of file will be filled in by KPR_TFTP_DOWNLOAD and KPR_TFTP_CALLBACK_DOWNLOAD, must be passed in by KPR_TFTP_UPLOAD. NB KPR_TFTP_CALLBACK_DOWNLOAD can send files of more than 64K, in which case this variable will have wrapped around back to $0000.

File Access Parameter Structure

Used By: KPR_FILE_LOAD

Offset	Size (bytes)	Contents
$00	$02	Pointer to ASCIIZ filename
$02	$02	For KPR_FILE_LOAD this field is a pointer to memory location data to be stored in (set this to $0000 when downloading a file where the first 2 bytes are the memory location the file should be stored in, e.g. a C64 PRG file)
$04	$02	Size of file will be filled in by KPR_FILE_LOAD
$06	$01	Device number: $00 – last accessed device (or default drive on first access) $01 – first drive on system (i.e. drive #8 on a C64) $02 – second drive on system (i.e. drive #9 on a C64) Etc

DNS Parameter Structure

Used By: KPR_DNS_RESOLVE

Offset	Size (bytes)	Contents
$00	$02	Pointer to asciiz hostname to resolve (can also be a dotted quad string)
$00	$04	IP address (filled in on successful resolution of hostname). If the same buffer is used for all KIPPER calls, then after a call to KPR_DNS_RESOLVE, the IP address should end up in the same memory location as the "Remote IP" field in the UDP Parameter Structure needed to call KPR_ SEND_UDP_PACKET

UDP Listener Parameter Structure

Used By: KPR_UDP_ADD_LISTENER

Offset	Size (bytes)	Contents
$00	$02	Port number to listen on (lo/high format)
$04	$02	Address of routine to be called when UDP packets arrive on specified port (lo/high format)

UDP/TCP Packet Parameter Structure

Used By: KPR_GET_INPUT_PACKET_INFO & KPR_SEND_UDP_PACKET

Offset	Size (bytes)	Contents
$00	$04	IP address of remote machine (source of inbound packets, destination of outbound packets)
$04	$02	Port number of remote machine (source of inbound packets, destination of outbound packets)
$06	$02	Port number of local machine (source of outbound packets, destination of inbound packets)
$08	$02	length of payload of packet (after all Ethernet, IP, UDP/TCP headers) in little-endian format. In a TCP connection, if the remote end terminates the connection, then the calling application will be notified via a packet having a length of $FFFF.
$0A	$02	Pointer to payload of packet.

TCP Connect Parameter Structure

Used By: KPR_TCP_CONNECT

Offset	Size (bytes)	Contents
$00	$04	IP address of remote machine. If this is 0.0.0.0, this creates a server connection, i.e. it will listen on the specified port for an inbound connection from a client. Set to any other value to initiate an outbound request (i.e. a client connection)
$04	$02	Port number – if this is a server connection, this port will be listened on. If this is a client connection then this is the remote port that will be connected to and a unique client port will be assigned.
$06	$02	Callback address – once the connection is made, the routine pointed at here will be called whenever new data arrives.

TCP Send Parameter Structure

Used By: KPR_SEND_TCP_PACKET

Offset	Size (bytes)	Contents
$00	$02	Payload length – length (in bytes) of data to be sent.
$02	$02	Payload pointer – pointer to data to be sent.

Block Copy Structure

Used By: KPR_BLOCK_COPY

Offset	Size (bytes)	Contents
$00	$02	Source address – pointer to first byte where block will be copied from
$02	$02	Destination address – pointer to first byte where block will be copied to
$04	$02	Block size – number of bytes to be copied

Error Codes

Code	Meaning
$80	Port in use
$81	Timeout on receive
$82	Transmit failed
$83	Transmission rejected by peer
$84	Input too large
$85	Device Failure
$86	Aborted by user
$87	Listener not available
$88	No such listener
$89	Connection reset by peer
$90	File Access Failure
$A0	Malformed URL
$A1	DNS Lookup Failed
$FE	Option not supported
$FF	Function not supported

Memory Map

Location	Contents
$A3..$B6	KIPPER page zero scratch area – programs should leave this area alone, and not assume that the use of any specific locations in this area will remain constant between versions of KIPPER
$334..$3FF	Additional scratch RAM area (this is the tape I/O buffer in the standard C64 memory map)
$2000..$2FFF	Used by the KIPPER web application server as temporary buffer (i.e. only used when calling KPR_HTTPD_START)
$8009..$800E	KIPPER signature – “KIPPER” (in hex, this is $4B $49 $50 $50 $45 $52)
$800F	KIPPER dispatch vector – JSR $800F to call any KIPPER function
$8012	KIPPER periodic processing vector – programs should call this location “regularly” to allow the IP stack to receive and process any inbound IP packets – at least a few times each second.
$8015.. $BFFF	KIPPER implementation code. Code in this region will vary between versions.
$C000..$CFFF	KIPPER scratch RAM – programs should not write to this area, nor should they assume variables in this space will remain constant between KIPPER implementations.
$DD0x	CIA #2 – KIPPER uses Timer A & Timer B on CIA #2. The timers are set up as a 16-bit counter, counting milliseconds.

Implementers Guide

This section contains information relevant for anyone wishing to create a cartridge that implements the KIPPER API, it may not be relevant for developers wishing to write programs that simply use that API.

To maximise compatibility:

- Make sure that if a call to one function fails, then a call to KPR_GET_LAST_ERROR will return a code indicating the type of error – this code should NOT be reset by successful function requests.
- Don't turn off interrupts for extended periods of time during any processing done during calls to KPR_PERIODIC_PROCESSING_VECTOR
- Keep any processing done by the routine called through KPR_VBL_VECTOR as short as possible.
- The output of KPR_PRINT_IP_CONFIG can be formatted in anyway you chose, and can include device specific information if appropriate. However the format of other KPR_PRINT_* functions should match that of the kipper implementation.
- The KPR_PRINT_* functions should use the kernal routine at $ffd2 to send output to the screen.

TFTP Directory Listings

A previous draft of the API had a function that used a proprietary tftp opcode to allow tftp directory listings. This function has now been removed, since using a proprietary opcode can prevent NAT from working correctly on the typical broadband routers most people are likely to have on their home networks.

So a new approach has been taken to allow directory listings without using proprietary opcodes. The protocol is not part of the KIPPER API itself, rather it is implemented solely within the kipper tftp server.

The kipper tftp server will treat any file download request where the filename starts with a $ as being a directory listing request. The remainder of the filename (after the leading $) is treated as a filemask. For example, doing a download request for a file named “$*.prg” will result in retrieving a list of all filenames that end with the extension .prg.

The resulting file will consist of null-terminated ASCII (not PETSCII) strings, with an extra null byte at the end of the last string in the listing.

So to use the kipper directory listing feature, the calling application must

1) construct an appropriate filename ($ followed by filemask).
2) Call **KPR_TFTP_DOWNLOAD** with a parameter block containing a pointer to the constructed filename, and the address of the buffer the directory listing will be placed in

Licenses

The KipperKart (i.e. the initial implementation of the KIPPER API) is licensed under the Mozilla Public License version 1.1 - http://www.mozilla.org/MPL/MPL-1.1.html

The KIPPER API specification (including this document) is released into the public domain.

This means:

- A program that *uses* the functions in the KIPPER API has no licensing restrictions imposed on it because of that usage.
- A program that *implements* the KIPPER API, but does not use any of the original KipperKart or ip65 source code in that implementation is also free to be licensed however the author of that implementation choses.
- A program that *reuses* any of the KipperKart of ip65 source code must be licensed under the terms of the MPL.