SunRay Protocol Documentation

© Paul Evans, 2003

`pe208@cam.ac.uk` or `nerd@freeuk.com`

Version 0.4.1

1 Introduction

Numbers in normal type (e.g 12) are in decimal. numbers in `fixed type` and prefixed with `0x` (e.g. `0x15`) are in hexadecimal.

1.1 Disclaimer

All information contained in this document was obtained by reverse engineering of packet dumps obtained from SunRay to server communications, and by noting effects on the SunRay of further communications started from code I have written. No Sun Microsystems code on either the server or the SunRay was decompiled or otherwise tampered with, nor any special modifications to the SunRay hardware were done in order to perform this work.

Further, this information is supplied in good faith but with no warranty of correctness or applicability for any purpose. Due to the nature of the reverse engineering performed, the reliability of this information cannot be guaranteed. I take no responsibility for any effects of applying information contained herein.

The names Sun Microsystems and SunRay are trademarks of the Sun Microsystems Inc.

2 Overview

The SunRay device has a fairly simple communications protocol, with a slightly more complex startup routine. The basic outline is that on powerup, it seeks configuration via DHCP, then authenticates with a server using TCP. Once this is set up, the server communicates with is via UDP. Each of the three protocols will be discussed here.

3 DHCP

The SunRay uses standard DHCP with some vendor-specific extensions. The vendor-specific parameters I have observed in use are:

number	type	name	description
21	ip address	authserver	TCP Authentication server IP address
22	uint 16	authport	TCP Authentication server port number
23	text	version	Firmware version string
24	ip address	logserver	Logging server IP address
25	uint 8	logkernel	Logging level for kernel messages
26	uint 8	lognet	Logging level for network messages
27	uint 8	logusb	Logging level for USB messages
28	uint 8	logvid	Logging level for video messages
29	uint 8	logappl	Logging level for application messages
31	ip address	fwserver	Unknown; possibly firmware server
32	uint 32	displayres	Initial display resolution
33	text	netname	Unknown

4 TCP

Having obtained the address and port number of the auth server using the DHCP system described above, the SunRay then attempts to connect to the auth port. Normally this is port 7009. The format of communications on this port is a simple ASCII format, starting with an ASCII heading name, followed by a space, then a number of "name=value" sections, each separated with spaces. The final section followed by a newline (`0x0a`) character, rather than a space.

The SunRay begins by sending a fairly long sequence to the server; there being two items of importance here. The "id" parameter contains the hardware MAC address of the SunRay, used as a unique identifier, and "tokenSeq" contains the number 1. I am unsure quite why this number is sent; perhaps it serves to identify this authentication request in the case that more than one is sent.

The server then responds with a "discInf" reply, setting "access=denied".

The SunRay then re-requests, including the parameters "pn" to indicate the UDP port number that will be used, and "state" being either "connected" or "disconnected".

The server then replies with "connInf", setting "tokenSeq" as the number saved above, and "access=allowed" until the SunRay sends a "state=connected" message. When it has done this, the included port number can be used to start communications by the UDP system.

It is important at this point that the TCP port remains open, or else the SunRay will reset back to the DHCP stage.

5 UDP

All UDP packets start with a common header:

offset	name	description
0x00	`seqnum`	Sequence number
0x02	`flags`	Flags
0x04	`atype`	AType field
0x06	`bdir`	BDir field
0x08	data	

Note that all multi-byte fields on the SunRay are stored in big endian format.

The `seqnum` field simply increments by one for each packet sent. The server and the SunRay maintain their own, independent numbering. It is at present unclear what happens if either end is supplied with a packet out of sequence. The sequence numbering starts from `0x0001`.

The third field, `atype` shows how my field naming scheme works. Since I do not know the official Sun naming scheme for these packet fields, I must construct my own. Apart from the sequence number and flags fields, all fields from here on begin with the letters A, B, C, etc... and follow with a description of the sort of field.

Here, it describes the type of the following data. Two distinct values have been observed here:

0x0000	AType0
0x0001	AType1

AType0 packets are only seen during starting up the SunRay. Once the device has been initialised, all drawing and status messages use AType1 packets. The format of the AType0 and AType1 packets are the same. The significance of this field is, as yet, unclear.

Finally, the `btype` field takes one of two values:

0x0000	Server to SunRay
0x07d0	SunRay to Server

These two directions take different formats. These are explained below.

5.1 Server to SunRay

This type of packet consists of a constant length header, followed by a collection of varying length "opcode" sections (some are constant length, some whose length varies with the data contained therein). The header is laid out as follows:

offset	name	description
0x00	`cdat`	Unknown
0x02	`ddat`	Unknown
0x04	`edat`	Unknown
0x06	`fdat`	Observed as always `0x0000`
0x08	opcode sections	

The data fields are nearly always `0x000f`, `0x000a`, `0x0010`, `0x0000`, but this is not always the case, and it is unclear what any of the fields mean. I have tested the SunRay with my own server, and it works if I fill all these fields with zero.

After these four standard fields, is a collection of opcode sections. Each section starts with a standard header, and is followed by data of variable length. The format of each section's data is explained below.

At this point, due to UDP packet length restrictions, the maximum length of the remaining data is `1440` octets. If there is any less data than this then the packet stands by itself. If it is precisely this long then this packet is part of a longer chain, and must be joined to the next packet. Note that the packet following this one will still contain all fields up to `fdat`. Also note that if the following packet contains `1440` octets then the packet following that must also be joined, and so on. Only when a packet of length less than this restriction arrives can the data can finally be interpreted.

offset	**name**	**description**
`0x00`	`gopcode`	Opcode
`0x01`	`hflags`	Flags
`0x02`	`iseq`	Sequence number
`0x04`	`x`	X coordinate
`0x06`	`y`	Y coordinate
`0x08`	`width`	Width of region
`0x0a`	`height`	Height of region
`0x0c`	opcode data	

The opcode's value can be one of the following:

`0xa1`	Unknown
`0xa2`	FillRect
`0xa3`	FillRectBitmap
`0xa4`	CopyRect
`0xa5`	SetRectBitmap
`0xa6`	SetRect
`0xa8`	SetMouseBound
`0xa9`	SetMousePointer
`0xaa`	SetMousePosition
`0xab`	SetKeyLock
`0xac`	Unknown
`0xad`	Unknown
`0xaf`	Pad
`0xd1`	Unknown
`0xd8`	Unknown

In most cases the `x`, `y`, `width` and `height` fields refer to the region of the screen that the operation affects. The exact semantics are mentioned in the following sections. The `x` and `y` fields are referred to as the command point, and the combination of all four fields are referred to as the command rectangle. The rectangle extends from (x,y) to (`x + width - 1`,`y + height - 1`) inclusive.

The `iseq` field is a sequence number like that in the main packet header. It starts at zero, and is incremented before sending each packet (i.e. so the first sequence number actually seen on the network would be `0x0001`). However, it is not incremented before sending a `0xaf gopcode` section. The full reasoning behind this is explained in the description of the `0xaf` opcode.

Unknown (`0xa1`)

The format of this packet is not known. It is seen only once, on startup. It is therefore difficult to make any assumptions about the data, but both occurrences I have seen the data has been 4 octets long. The command rectangle's definition was `0x0000`, `0x0000`, `0x0001`, `0x0001`.

offset	name	description
`0x00`	`unknown`	4 octets of unknown data
`0x04`	next opcode	

FillRect (`0xa2`)

This drawing operation fills the command rectangle in a solid colour.

offset	name	description
`0x00`	`colour`	Colour to set rectangle
`0x04`	next opcode	

FillRectBitmap (`0xa3`)

This drawing operation fills the command rectangle in a colour, given by bitmap data. This allows masked filling of rectangles and is primarily used for setting text characters on the screen.

offset	name	description
`0x00`	`colour`	Colour to set rectangle
`0x04`	`bitmap`	Bitmap field
`n`	next opcode	

The bitmap data itself is a sequence of `1`'s and `0`'s. The data for the first screen line (i.e. of y value given in the header) is given first, starting at the initial `x` coordinate and increasing. This data is zero-padded up to the nearest octet boundary, then the next line is given. This continues for all screen lines. Finally the bitmap is zero-padded to the nearest 32bit boundary, for aligning the next opcode.

This means that, given a function `roundup(n,a)` which returns the value of `n` rounded up to the nearest multiple `a`, the length of the bitmap field is

```
roundup((roundup(width, 8) / 8) * height, 4)
```

octets.

CopyRect (`0xa4`)

This drawing operation copies part of the screen into the command rectangle. The command and "source" rectangle can overlap; the copy operation will be done to ensure the copy is correct. There is no limit on the size of the copy operation - the whole screen can be copied; for example, to implement whole-screen scrolling.

offset	**name**	**description**
`0x00`	`xsrc`	X coordinate of source
`0x02`	`ysrc`	Y coordinate of source
`0x04`	next opcode	

SetRectBitmap (`0xa5`)

This drawing operation sets the command rectangle of the screen to two colours, given by bitmap data. This allows filling rectangles primarily used for text characters on the screen, or other monotone screen graphics.

offset	**name**	**description**
`0x00`	`colour0`	Colour to set rectangle for `0` bits
`0x04`	`colour1`	Colour to set rectangle for `1` bits
`0x08`	`bitmap`	Bitmap field
`n`	next opcode	

The rectangle and bitmap data are in similar format as above. The difference between this opcode and FillRectBitmap is that here, two colours are specified and are drawn depending in the bit in the bitmap, whereas for FillRectBitmap, the screen is left unmodified for the `0` bits.

SetRect (`0xa6`)

This drawing operation sets the command rectangle to arbitrary colour data as specified in the data.

offset	**name**	**description**
`0x00`	`colours`	Colours to set rectangle
`n`	next opcode	

The colour data is specified as a long list of 3 octet colour values, in the same order as for the bitmaps above (data for the first row first, starting at the initial coordinate).

SetMouseBound (`0xa8`)

This operation sets the boundary of the mouse cursor's allowed movement.

offset	name	description
0x00	xbound	Low X coordinate of bounds
0x02	ybound	Low Y coordinate of bounds
0x04	wbound	Width of bounds
0x06	hbound	Height of bounds
0x08	next opcode	

The four parameters above define the mouse bound rectangle.

I have observed an odd function of the command rectangle in this command. The starting position does not seem to be significant, though the size of it does. If the size is set smaller than the screen resolution, drawing operations get clipped to this size, and if the mouse falls outside then it disappears permanently from the screen (i.e. does not reappear when moved back again).

SetMouseCursor (`0xa9`)

This operation sets the shape and colour of the mouse pointer. It does not directly affect the screen's framebuffer display. The command point gives the coordinates inside the bitmap of the mouse cursor's hotpoint, and the size of the command rectangle gives the size of the cursor.

offset	name	description
0x00	colourf	Colour to set foreground
0x04	colourg	Colour to set background
0x08	bitmapc	Colour bitmap field
n1	bitmapm	Mask bitmap field
n2	next opcode	

Note that two fields of bitmap data are present here; so the formula quoted above for the length of bitmap fields will need to be applied twice. The colour and mask bitmaps are of similar format to those above, and work together as follows:

mask	colour	description
0	X	transparent
1	0	background
1	1	foreground

SetMousePosition (`0xaa`)

This operation sets the mouse cursor position, and is normally used only during startup. The command point gives the required cursor position, the size of the command rectangle is set to zero.

offset	name	description
0x00	padding	4 octets of zero-padding
0x04	next opcode	

SetKeyLock (`0xab`)

Here the command rectangle does not give any useful data, and so is set to all zero.

offset	name	description
0x00	llock	Keyboard lock value
0x02	mdat	The value 0x0000
0x04	next opcode	

The llock field is an inclusive OR of the required LEDs on the keyboard.

mask	lock
0x0001	Number lock
0x0002	Caps lock

Having been unable to activate either Scroll lock or Compose, I am unable to say what values these would be, but I suspect 0x0004 and 0x0008 would possibly be the correct values.

Alternatively, these bitmasks could be part of USB's Keyboard HID values, as the only two known cases (above) match the USB specifications.

Unknown (0xac)

The format of this opcode is not known, other than it is of constant length. The command rectangle is all zero.

offset	name	description
0x00	ldat	Unknown data
0x04	mdat	Unknown data
0x08	next opcode	

I have observed that all occurrences of this packet contain an all-zero command rectangle, ldat as constant 1, and mdat as a small odd number (1, 3, 5, 7 or 9). Sometimes the packet occurs multiple times

Unknown (0xad)

The format of this opcode is not known, other than how to calculate its length. The size of the command rectangle does not seem to directly relate to the size of the data.

offset	name	description
0x00	len	Hint to length of variable data
0x02	dat	Data of unknown format
n	next opcode	

The length of the dat field is given by (len & 0xfffc) + 2.

Pad (0xaf)

The contents of this command are observed always constant. The command rectangle fields are (respectively) 0x0000, 0x0001, 0xffff and 0xffff, and the remaining data as below:

offset	name	description
0x00	ndat	4 octets of 0xff
0x04	next opcode	

This packet serves as marker for the current `iseq` value. The value associated with this packet is the value that the server last sent; but the value itself is not incremented. The SunRay can then confirm if it has received all previous commands, and it not; it can issue a NACK reply (see below).

Unknown (`0xd1`)

This packet type is only seen once; on startup. While the purpose of this packet is not known, the contents are observed always constant. The command rectangle fields are all zero, and there is no other data.

offset	name	description
0x00	next opcode	

This packet is seen only once, during startup, and it occurs on its own. I.e. it is the only opcode type within the UDP packet. Before this packet, all packets from the server are of the AType0 variety. Afterwards, they are AType1. I suspect this packet therefore signifies that the startup routines are complete, and that the SunRay should now enter normal operation.

Unknown (`0xd8`)

This packet is seen occasionally while the sunray is running. After the server sends this packet, the SunRay and server perform some additional communication using a new UDP port number at the server end, then communication continues as normal via this new port. I suspect this is because the server may need the UDP port for other operations, so renegotiates the port.

offset	name	description
0x00	next opcode	

5.2 SunRay to Server

The messages from the SunRay to the server follow a similar scheme to that above; a constant sized header then a string of opcode sections.

offset	name	description
0x00	cdat	Unknown
0x02	ddat	Unknown
0x04	edat	Observed as always 0x0000
0x06	fdat	Observed as always 0x0000
0x08	opcode sections	

I am unsure of what `cdat` and `ddat` are for, but their value doesn't seem to change over small times. I.e. it stays at one value for a while, and changes to a different value after a few seconds. The values do not seem significant to any part of the data contained within the opcode section.

As before, a sequence of opcode sections follow this header. Unlike as above, there is no packet joining mechanism, as the replies from the SunRay tend to be much shorter. Note also that, whereas the server's messages need at least one opcode field, SunRay's replies quite often contain no additional opcodes; therefore the packet ends after the header.

All the opcodes start with the following opcode header:

offset	name	description
`0x00`	`gopcode`	Opcode
`0x01`	`hdat`	Unknown
`0x03`	`idat`	Unknown
`0x04`	opcode data	

The `gopcode` field plays a similar role to that in the server's packets, but unlike in the server's packets, there appears to be no sequence number field. Neither `hdat` nor `idat` seem to follow any particular pattern; their value does not seem to relate to the data in the packet.

The following `gopcode`s have been observed.

`0xc1`	Keyboard
`0xc2`	Mouse
`0xc4`	NACK
`0xc5`	Unknown
`0xc6`	Unknown
`0xc7`	Unknown

Keyboard (`0xc1`)

This packet indicates a change of state on the keyboard (a key has been pressed or released).

offset	name	description
`0x00`	`jdat`	Constant `0x0021`
`0x02`	`kshift`	Shift value
`0x04`	`lkeys`	Keyboard scan codes
`0x0a`	`mdat`	Constant `0x00`
`0x0c`	next opcode	

The keyboard scan codes are taken from the USB keyboard HID specifications. For example, the 26 letters of the Roman alphabet are assigned the scan codes `0x04` to `0x1d` in alphabetical order. `lkeys` is an 6-element array of scancodes. The array is filled from the bottom first, new keys are added at the top, when a key is released it is removed from the list and the remaining keys are moved down. Therefore, the array always contains some scan codes (or possibly none), followed by `0x00` padding.

An interesting effect is noticed if more than three keys are held at once. If this is the case, all 6 entries in the array become `0x01`, and remain there until not more than three keys are depressed. This is clearly an error-type code, signifying that there are too many keys pressed, however I am unsure why it should limit to three keys, given that there are six spaces in the array.

The shift values are an inclusive OR from the following:

mask	key
`0x0001`	Control
`0x0002`	Left shift
`0x0004`	Alt
`0x0008`	Left Sun
`0x0010`	Compose
`0x0020`	Right shift
`0x0040`	Alt Graph
`0x0080`	Right Sun
`0x0100`	Num lock
`0x0200`	Caps lock

Note that the shift values of Num lock and Caps lock refer to the state of the lock being set (if the LED is on), and not if the key itself is depressed.

Mouse (`0xc2`)

This opcode specifies the position and state of the mouse.

offset	name	description
`0x00`	`jbuttons`	State of the buttons
`0x02`	`kx`	X coordinate of pointer
`0x04`	`ly`	Y coordinate of pointer
`0x06`	`mdat`	Constant `0x0000`
`0x08`	next opcode	

The `jbuttons` field has a bit set for each of the first three buttons. If the mouse has more than three buttons they are ignored. This field also has bit 6 (`0x0040`) set, but I do not know why.

NACK (`0xc4`)

I have observed this error as a response to incorrect `iseq` values being sent in the packet chain. More investigation, using purposely-malformed sequences will be needed to determine its exact nature.

offset	name	description
`0x00`	`jdat`	Unknown data
`0x04`	`kmin`	Unknown data; suspected sequence number
`0x08`	`lmax`	Unknown data; suspected sequence number
`0x0c`	next opcode	

I have observed that `jdat` is always 1 in all the packets I have seen. These packets concerned incorrect `iseq` numbers; but perhaps the main packet header `seqnum` can also cause error messages. During the malformed sequence; a packet containing `iseq=0x0058` was sent; following this was a packet containing `iseq=0x005b`.

The SunRay responded by sending `jdat=0x0001`, `kmin=0x0059`, `kmax=0x005a`. This would appear to indicate the missing `iseq` numbers. This appears to give the motivation for the behaviour of the `iseq` value when sending a `0xaf` command.

Unknown (`0xc5`)

The purpose of this packet is unknown.

offset	**name**	**description**
`0x00`	`jdat`	Unknown data
`0x04`	next opcode	

This packet type is only seen during startup; where a collection of seven similar packets are sent. The first octet of the data is always a valid `gopcode` for server-to-SunRay packets. They are always sent in the order `0xa6`, `0xa3`, `0xa5`, `0xa4`, `0xa2`, `0xa7`, `0xad`. The remaining three octets are different between `gopcodes`, but the same number is always sent at different startups for the same `gopcode`. I suspect these are either startup self-test timings, code versions, or similar.

Unknown (`0xc6`)

The exact purpose of this packet is unknown, but it contains, in ASCII, the name of the firmware version, as sent in the DHCP communication during startup. As above, this packet is only seen during startup, though it does appear multiple times; almost identically. Only a few octets of data change from repeat to repeat, but I do not know the significance of what changes.

offset	**name**	**description**
`0x00`	`jlen`	Data length
`0x02`	`kstrlen`	String length
`0x03`	`lstring`	String data
`n1`	`mdat`	Unknown data
`n2`	next opcode	

The total length of the data in this packet is given in `jlen`; this length including the length of the `jlen` field itself. Following then is the firmware string, with the length before it. After this is some other data, whose format is as yet unknown. It does not appear to be significant.

Unknown (`0xc7`)

This packet appears to contain three sets of rectangle definitions. It is sent along with `0xc1` and `0xc2` every time the SunRay sends status information back to the server.

offset	**name**	**description**
`0x00`	`x1`	X1 coordinate
`0x02`	`y1`	Y1 coordinate
`0x04`	`width1`	Width1
`0x06`	`height1`	Height1
`0x08`	`x2`	X2 coordinate
`0x0a`	`y2`	Y2 coordinate
`0x0c`	`width2`	Width2
`0x0e`	`height2`	Height2
`0x10`	`x3`	X3 coordinate
`0x12`	`y3`	Y3 coordinate
`0x14`	`width3`	Width3
`0x16`	`height3`	Height3
`0x18`	next opcode	

I am unsure of what these three rectangles mean, because so far I have observed that each one is always (0,0)-(1279,1023). I suspect they may have something to do with real and virtual screen resolutions.