NAME

srec_examples − examples of how to use SRecord

DESCRIPTION

The *srec_cat* command is very powerful, due to the ability to combine the the input filters in almost unlimited ways. This manual page describes a few of them.

This manual page describes how to use the various input files, input filters and input generators. But these are only examples, for more complete details, see the *srec_input*(1) manual page.

The Commands Lines Are Too Long

If you are marooned on an operating system with absurdly short command line length limits, some of the commands which follow may be too long. You can get around this handicap by placing your command line in a file, say *fred.txt*, and then tell *srec_cat*(1) to read this file for the rest of its command line, like this

```
srec_cat @fred.txt
```

This also has the advantage of allowing comments, several lines and even indenting to make it more clear. Comments start at a "#" and extend to the end of the line. Blank lines are ignored.

Of course, you could always upgrade to Linux, which has been sucking less for over 17 years now.

Your Examples Wanted

If you have a clever way of using SRecord, or have solved a difficult problem with SRecord, you could contribute to this manual page, making it more useful for everyone. Send your contribution in an email to the email address at the end of this manual page.

CONVERTING FILE FORMATS

The simplest of the things *srec_cat*(1) can do is convert from one EPROM file format to another. Please keep in mind, as you read this section, that you can do many of these things simultaneously in one command. They are only broken out separately to make them easier to understand.

Intel to Motorola

One of the simplest examples is converting files from Intel hex format to Motorola S-Record format:

```
srec_cat intel-file -intel -o srec-file
```

Pick any two formats that SRecord understands, it can convert between all of them. (Except the assembler, BASIC, C and FPGA outputs which are write only.)

Motorola to Intel

Converting the other way is just as simple:

```
srec_cat srec-file -o intel-file -intel
```

The default format is Motorola S-Record format, so it does not need to be specified.

Different Shapes of the Same Format

It is regrettably common that some addle-pated EPROM programmers only implement a portion of the specification used to represent their hex files. For example, some compilers produce "s19" Motorola data (that is, S1 data records with S9 start records, 16 bit address fields) which would be OK except that some blockhead EPROM programmers insist on "s37" Motorola data (that is, S3 data records with S7 start records, 32 bit address fields).

It is possible to convert from one Motorola shape to another using the **−Address-Length** option:

```
srec_cat short.srec -o long.srec --address-length=4
```

This command says to use four byte (32-bit) addresses on output.

This section also applies to Intel hex files, as they, too, have the ability to select from a variety of address widths.

Line Lengths

From time to time you will come across a feeble-minded EPROM programmer that can't cope with long SRecord lines, they assume that there will only ever be 16 bytes of data per line, and barf when they see the default 32 byte payloads that *srec_cat*(1) writes.

The Motorola S-record format definition permits up to 255 bytes of payload. All EPROM programmers *should* have sufficiently large buffers to cope with records this big. Few do.

The −line-length option may be used to specify the maximum line length (not including the newline) to be used on output. For example, 16 byte payloads for Motorola hex

```
srec_cat long.srec -o short.s19 --line-length=46
```

The line length option interacts with the address length option, so some tinkering to optimize for your particular situation many be necessary.

Just the Data, Please

There are some bonehead EPROM programmers which can only cope with data records, and are unable to cope with header records or execution start address records. If you have this problem, the **−data-only** option can be used to suppress just about everything except the data. The actual effect depends on the format, of course, because some don't have these features anyway.

The **−data-only** option is short hand. There are four properties which may be **−disabled** or **−enabled** separately. See the *srec_cat*(1) man page for a description of the **−disabled** and **−enabled** options.

For example, your neanderthal EPROM programmer requires Motorola hex with header records (S0), but without data count (S5) records. Not using the **−data-only** option has it barf on the data count record, but using the **−data-only** option has it barf on the missing header record. Using the **−disable=data-count** option would leave the header record intact while supressing the data count record.

Data Headers

The *srec_cat*(1) command always tries to pass through header records unchanged, whenever they are present. It even tries preserve them across file format changes, to the limit the file formats are capable of.

If there is no file header record and you would like to add one, or you wish to override an existing file header record, use the **−header**=*string* option. You will need to quote the string (to insulate it from the shell) if it contains spaces or shell meta-characters.

Execution Start Addresses

The *srec_cat*(1) command always tries to pass through execution start addresses (typically occurring at the end of the file), whenever they are present. They are adjusted along with the data records by the **−offset** filter. It even tries preserve them across file format changes, to the limit the file formats are capable of.

If there is no execution start address record and you would like to add one, or you wish to override an existing execution start address record, use the **−execution-start-address**=*number* option.

Please note: the execution start address is a different concept than the first address in memory of your data. Think of it as a "goto" address to be jumped to by the monitor when the hex load is complete. If you want to change where your data starts in memory, use the **−offset** filter.

Fixing Checksums

Some embedded firmware developers are saddled with featherbrained tools which produce incorrect checksums, which the more vigilant models of EPROM programmer will not accept.

To fix the checksums on a file, use the **−ignore-checksums** option. For example:

```
srec_cat broken.srec --ignore-checksums -o fixed.srec
```

The checksums in *broken.srec* are parsed (it is still and error if they are absent) but are not checked. The resulting *fixed.srec* file has correct checksums. The **−ignore-checksums** option only applies to input.

This option may be used on any file format which has checksums, including Intel hex.

Discovering Mystery Formats

See the **What Format Is This?** section, below, for how to discover and convert mystery EPROM load file formats.

BINARY FILES

It is possible to convert to and from binary files. You can even mix binary files and other formats together in the same *srec_cat*(1) command.

Writing Binary Files

The simplest way of reading a hex file and converting it to a binary file looks like this:

```
srec_cat fred.hex -o fred.bin -binary
```

This reads the Mororola hex file *fred.srec* and writes it out to the *fred.bin* as raw binary.

Note that the data is placed into the binary file at the byte offset specified by the addresses in the hex file. If there are holes in the data they are filled with zero. This is, of course, common with linker output where the code is placed starting at a particular place in memory. For example, when you have an image that starts at 0x100000, the first 1MB of the output binary file will be zero.

You can automatically cancel this offset using a command like

```
srec_cat fred.hex -offset - -minimum-addr fred.hex -o fred.bin
```

The above command works by offseting the *fred.hex* file lower in memory by the least address in the *fred.hex* file's data.

See also the *srec_binary*(5) man page for additional detail.

Reading Binary Files

The simplest way of reading a binary file and converting it looks like this

```
srec_cat fred.bin -binary -o fred.srec
```

This reads the binary file *fred.bin* and writes all of its data back out again as a Motorola S-Record file.

Often, this binary isn't exactly where you want it in the address space, because it is assumed to reside at address zero. If you need to move it around use the **−offset** filter.

```
srec_cat fred.bin -binary -offset 0x10000 -o fred.srec
```

You also need to avoid file "holes" which are filled with zero. You can use the **−crop** filter, of you could use the **−unfill** filter if you done't know exactly where the data is.

```
srec_cat fred.bin -binary -unfill 0x00 512 -o fred.srec
```

The above command removes runs of zero bytes that are 512 bytes long or longer. If your file contains 1GB of leading zero bytes, this is going to be slow, it may be better to use the *dd*(1) command to slice and dice first.

JOINING FILES TOGETHER

The *srec_cat* command takes its name from the UNIX *cat*(1) command, which is short for "catenate" or "to join". The *srec_cat* command joins EPROM load files together.

All In One

Joining EPROM load files together into a single file is simple, just name as many files on the command line as you need:

```
srec_cat infile1 infile2 -o outfile
```

This example is all Motorola S-Record files, because that's the default format. You can have multiple formats in the one command, and *srec_cat*(1) will still work. You don't even have to output the same format:

```
srec_cat infile1 -spectrum infile2 -needham \
    -o outfile -signetics
```

These are all ancient formats, however it isn't uncommon to have to mix and match Intel and Motorola formats in the one project.

Filtering After Joining

There are times when you want to join two sets of data together, and then apply a filter to the joined result. To do this you use parentheses.

```
srec_cat                                                    \
    '('                                                     \
        infile --exclude 0xFFF0 0x10000                     \
        --generate 0xFFF0 0xFFF8 --repeat-string 'Bananas ' \
    ')'                                                     \
    --b-e-length 0xFFF8 4                                   \
    --b-e-checksum-neg 0xFFFC 4 4                           \
    -o outfile
```

The above example command catenates an input file (with the generated data area excluded) with a constant string. This catenated input is then filtered to add a 4-byte length, and a 4-byte checksum.

Joining End-to-End

All too often the address ranges in the EPROM load files will overlap. You will get an error if they do. If both files start from address zero, because each goes into a separate EPROM, you may need to use the

offset filter:

```
srec_cat infile1 \
    infile2 -offset 0x80000 \
    -o outfile
```

Sometimes you want the two files to follow each other exactly, but you don't know the offset in advance:

```
srec_cat infile1 \
    infile2 -offset -maximum-addr infile1 \
    -o outfile
```

Notice that where the was a number (0x80000) before, there is now a calculation (−maximum-addr *infile1*). This is possible most places a number may be used (also −minimum-addr and −range).

CROPPING THE DATA

It is possible to copy an EPROM load file, selecting addresses to keep and addresses to discard.

What To Keep

A common activity is to crop your data to match your EPROM location. Your linker may add other junk that you are not interested in, *e.g.* at the RAM location. In this example, there is a 1MB EPROM at the 2MB boundary:

```
srec_cat infile -crop 0x200000 0x300000 \
    -o outfile
```

The lower bound for all address ranges is inclusive, the upper bound is exclusive. If you subtract them, you get the number of bytes.

Address Offset

Just possibly, you have a moronic EPROM programmer, and it barfs if the EPROM image doesn't start at zero. To find out just where is *does* start in memory, use the *srec_inf*(1) command:

```
$ srec_info example.srec
Format: Motorola S-Record
Header: extra-whizz tool chain linker
Execution Start Address: 0x00200000
Data:   0x200000 - 0x32AAEF
$
```

Rather than butcher the linker command file, just offset the addresses:

```
srec_cat infile -crop 0x200000 0x300000 -offset -0x200000 \
    -o outfile
```

Note that the offset given is *negative*, it has the effect of subtracting that value from all addresses in the input records, to form the output record addresses. In this case, shifting the image back to zero.

This example also demonstrates how the input filters may be chained together: first the crop and then the offset, all in one command, without the need for temporary files.

If all you want to do is offest the data to start from address zero, this can be automated, so you don't have to know the minimum address in advance, by using *srec_cat*'s ability to calculate some things on the command line:

```
srec_cat infile -offset - -minimum infile \
    -o outfile
```

Note the spaces either side of the minus sign, they are mandatory.

What To Throw Away

There are times when you need to exclude an small address range from an EPROM load file, rather than wanting to keep a small address range. The **−exclude** filter may be used for this purpose.

For example, if you wish to exclude the address range where the serial number of an embedded device is kept, say 0x20 bytes at 0x100, you would use a command like this:

```
srec_cat input.srec -exclude 0x100 0x120 -o output.srec
```

The *output.srec* file will have a hole in the data at the necessary locations.

Note that you can have both **−crop** and **−exclude** on the same command line, whichever works more naturally for your situation.

Discontinuous Address Ranges

Address ranges don't have to be a single range, you can build up an address range using more than a single pair.

```
srec_cat infile -crop 0x100 0x200 0x1000 0x1200 \
    -o outfile
```

This filter results in data from 0x100..0x1FF and data from 0x1000..0x1200 to pass through, the rest is dropped. This is is more efficient than chaining a −crop and an −exclude filter together.

MOVING THINGS AROUND

It is also possible to change the address of data records, both forwards and backwards. It is also possible rearrange where data records are placed in memory.

Offset Filter

The **−offset=***number* filter operates on the addresses of records. If the number is positive the addresses move that many bytes higher in memory, negative values move lower.

```
srec_cat infile -crop 0x200000 0x300000 -offset -0x200000 \
    -o outfile
```

The above example moves the 1MB block of data at 0x200000 down to zero (the offset is *negative*) and discards the rest of the data.

Byte Swapping

There are times when the bytes in the data need to be swapped, converting between big-endian and little-endian data usually.

```
srec_cat infile --byte-swap 4 -o outfile
```

This reverses bytes in 32 bit values (4 bytes). The default, if you don't supply a width, is to reverse bytes in 16 bit values (2 bytes). You can actually use any weird value you like, although 64 bits (8 bytes) may be useful one day.

Binary Output

You need to watch out for binary files on output, because the holes are filled with zeros. Your 100kB program at the top of 32-bit addressed memory will make a 4GB file. See *srec_binary*(5) for how understand and avoid this problem, usually with the −offset filter.

Splitting an Image

If you have a 16-bit data bus, but you are using two 8-bit EPROMs to hold your firmware, you can generate the even and odd images by using the −SPlit filter. Assuming your firmware is in the *firmware.hex* file, use the following:

```
srec_cat firmware.hex -split 2 0 -o firmware.even.hex
srec_cat firmware.hex -split 2 1 -o firmware.odd.hex
```

This will result in the two necessary EPROM images. Note that the output addresses are divided by the split multiple, so if your EPROM images are at a particular offset (say 0x10000, in the following example), you need to remove the offset, and then replace it...

```
srec_cat firmware.hex \
    -offset -0x10000 -split 2 0 \
    -offset 0x10000 -o firmware.even.hex
srec_cat firmware.hex \
    -offset -0x10000 -split 2 1 \
    -offset 0x10000 -o firmware.odd.hex
```

Note how the ability to apply multiple filters simplifies what would otherwise be a much longer script.

Striping

A second use for the −SPlit filter is memory striping. In this example, the hardware requires that 512-byte blocks alternate between 4 EPROMs. Generating the 4 images would be done as follows:

```
srec_cat firmware.hex -split 0x800 0x000 0x200 -o firmware.0.hex
srec_cat firmware.hex -split 0x800 0x200 0x200 -o firmware.1.hex
srec_cat firmware.hex -split 0x800 0x400 0x200 -o firmware.2.hex
srec_cat firmware.hex -split 0x800 0x600 0x200 -o firmware.3.hex
```

Unspliting Images

The unsplit filter may be used to reverse the effects of the split filter. Note that the address range is expanded leaving holes between the stripes. By using all the stripes, the complete input is reassembled, without any holes.

```
srec_cat -o firmware.hex \
    firmware.even.hex -unsplit 2 0 \
    firmware.odd.hex  -unsplit 2 1
```

The above example reverses the previous 16-bit data bus example,.

FILLING THE BLANKS

Often EPROM load files will have “holes” in them, places where the compiler and linker did not put anything. For some purposes this is OK, and for other purposes something has to be done about the holes.

The Fill Filter

It is possible to fill the blanks where your data does not lie. The simplest example of this fills the entire EPROM:

```
srec_cat infile -fill 0x00 0x200000 0x300000 -o outfile
```

This example fills the holes, if any, with zeros. You must specify a range − with a 32-bit address space, filling everything generates *huge* load files.

If you only want to fill the gaps in your data, and don’t want to fill the entire EPROM, try:

```
srec_cat infile -fill 0x00 -over infile -o outfile
```

This example demonstrates the fact that wherever an address range may be specified, the **−over** and **−within** options may be used.

Unfilling the Blanks

It is common to need to “unfill” an EPROM image after you read it out of a chip. Usually, it will have had all the holes filled with 0xFF (areas of the EPROM you don’t program show as 0xFF when you read them back).

To get rid of all the 0xFF bytes in the data, use this filter:

```
srec_cat infile -unfill 0xFF -o outfile
```

This will get rid of *all* the 0xFF bytes, including the ones you actually wanted in there. There are two ways to deal with this. First, you can specify a minimum run length to the un-fill:

```
srec_cat infile -unfill 0xFF 5 -o outfile
```

This says that runs of 1 to 4 bytes of 0xFF are OK, and that a hole should only be created for runs of 5 or more 0xFF bytes in a row. The second method is to re-fill over the intermediate gaps:

```
srec_cat outfile -fill 0xFF -over outfile \
    -o outfile2
```

Which method you choose depends on your needs, and the shape of the data in your EPROM. You may need to combine both techniques.

Address Range Padding

Some data formats are 16 bits wide, and automatically fill with 0xFF bytes if it is necessary to fill out the other half of a word which is not in the data. If you need to fill with a different value, you can use a command like this:

```
srec_cat infile -fill 0x0A \
    -within infile -range-padding 2 \
    -o outfile
```

This gives the fill filter an address range calculated from details of the input file. The address range is all the address ranges covered by data in the *infile*, extended downwards (if necessary) at the start of each sub-range to a 2 byte multiple and extended upwards (if necessary) at the end of each sub-range to a 2 byte multiple. This also works for larger multiples, like 1kB page boundaries of flash chips. This address range padding works anywhere an address range is required.

Fill with Copyright

It is possible to fill unused portions of your EPROM with a repeating copyright message. Anyone trying to reverse engineer your EPROMs is going to see the copyright notice in their hex editor.

This is accomplished with two input sources, one from a data file, and one which is generated on-the-fly.

```
srec_cat infile \
    -generate '(' 0 0x100000 -minus -within infile ')' \
        -repeat-string 'Copyright (C) 1812 Tchaikovsky.  ' \
    -o outfile
```

Notice how the address range for the data generation: it takes the address range of your EPROM, in this case 1MB starting from 0, and subtracts from it the address ranges used by the input file.

If you want to script this with the current year (because 1812 is a bit out of date) use the shell's output substitution (back ticks) ability:

```
srec_cat infile \
    -generate '(' 0 0x100000 -minus -within infile ')' \
        -repeat-string "Copyright (C) `date +%Y` Tchaikovsky.  " \
    -o outfile
```

The string specified is repeated over and over again, until it has filled all the holes.

Obfuscating with Noise

Sometimes you want to fill your EPROM images with noise, to conceal where the real data stops and starts. You can do this with the **−random-fill** filter.

```
srec_cat infile -random-fill 0x200000 0x300000 \
    -o outfile
```

It works just like the **−fill** filter, but uses random numbers instead of a constant byte value.

Fill With 16-bit Words

When filling the image with a constant byte value doesn't work, and you need a constant 16-bit word value instead, use the **−repeat-data** generator, which takes an arbitrarily long sequence of bytes to use as the fill pattern:

```
srec_cat infile \
    -generator '(' 0x200000 0x300000 -minus -within infile ')' \
        -repeat-data 0x1B 0x08 \
    -o outfile
```

Notice how the generator's address range once again avoids the address ranges occupied by the *infile*'s data. You have to get the endian-ness right yourself.

INSERTING CONSTANT DATA

From time to time you will want to insert constant data, or data not produced by your compiler or assembler, into your EPROM load images.

Binary Means Literal

One simple way is to have the desired information in a file. To insert the file's contents literally, with no format interpretation, use the *binary* input format:

```
srec_cat infile --binary -o outfile
```

It will probably be necessary to use and *offset* filter to move the data to where you actually want it within the image:

```
srec_cat infile --binary --offset 0x1234 -o outfile
```

It is also possible to use the standard input as a data source, which lends itself to being scripted. For example, to insert the current data and time into an EPROM load file, you could use a pipe:

```
date | srec_cat - -bin --offset 0xFFE3 -o outfile
```

The special file name "`-`" means to read from the standard input. The output of the *date* command is always 29 characters long, and the offset shown will place it at the top of a 64KB EPROM image.

Repeating Once

The **Fill with Copyright** section, above, shows how to repeat a string over and over. We can use a single repeat to insert a string just once.

```
srec_cat -generate 0xFFE3 0x10000 -repeat-string "`date`" \
    -o outfile
```

Notice how the address range for the data generation exactly matches the length of the *date*(1) output size. You can, of course, add your input file to the above *srec_cat*(1) command to catenate your EPROM image

together with the date and time.

DATA ABOUT THE DATA

It is possible to add a variety of data about the data to the output.

Checksums

The **−big-endian-checksum-negative** filter may be used to sum the data, and then insert the negative of the sum into the data. This has the effect of summing to zero when the checksum itself is summed across, provided the sum width matches the inserted value width.

```
srec_cat infile \
        --crop 0 0xFFFFFC \
        --random-fill 0 0xFFFFFC \
        --b-e-checksum-neg 0xFFFFFC 4 4 \
    -o outfile
```

In this example, we have an EPROM in the lowest megabyte of memory. The −crop filter ensures we are only summing the data within the EPROM, and not anywhere else. The −random-fill filter fills any holes left in the data with random values. Finally, the −b-e-checksum-neg filter inserts a 32 bit (4 byte) checksum in big-endian format in the last 4 bytes of the EPROM image. Naturally, there is a little endian version of this filter as well.

Your embedded code can check the EPROM using C code similar to the following:

```
unsigned long *begin = (unsigned long *)0;
unsigned long *end = (unsigned long *)0x100000;
unsigned long sum = 0;
while (begin < end)
    sum += *begin++;
if (sum != 0)
{
    Oops
}
```

The −big-endian-checksum-bitnot filter is similar, except that summing over the checksum should yield a value of all-one-bits (-1). For example, using shorts rather than longs:

```
srec_cat infile \
        --crop 0 0xFFFFFE \
        --fill 0xCC 0x00000 0xFFFFFE \
        --b-e-checksum-neg 0xFFFFFE 2 2 \
    -o outfile
```

Assuming you chose the correct endian-ness filter, your embedded code can check the EPROM using C code similar to the following:

```
unsigned short *begin = (unsigned long *)0;
unsigned short *end = (unsigned long *)0x100000;
unsigned short sum = 0;
while (begin < end)
    sum += *begin++;
if (sum != 0xFFFF)
{
    Oops
}
```

There is also a −b-e-checksum-positive filter, and a matching little-endian filter, which inserts the simple sum, and which would be checked in C using an equality test.

```
srec_cat infile \
        --crop 0 0xFFFFFF \
        --fill 0x00 0x00000 0xFFFFFF \
        --b-e-checksum-neg 0xFFFFFF 1 1 \
    -o outfile
```

Assuming you chose the correct endian-ness filter, your embedded code can check the EPROM using C code similar to the following:

```
unsigned char *begin = (unsigned long *)0;
unsigned char *end = (unsigned long *)0xFFFFF;
unsigned char sum = 0;
while (begin < end)
    sum += *begin++;
if (sum != *end)
{
    Oops
}
```

In the 8-bit case, it doesn't matter whether you use the big-endian or little-endian filter.

You can look at the checksum of your data, by using the "hex-dump" output format.

```
srec_cat infile \
        --crop 0 0x10000 \
        --fill 0xFF 0x0000 0x10000 \
        --b-e-checksum-neg 0x10000 4 \
        --crop 0x10000 0x10004 \
    -o - --hex-dump
```

This command reads in the file, checksums the data and places the checksum at 0x10000, crops the result to contain only the checksum, and then prints the checksum on the standard output in a classical hexadecimal dump format.

Cyclic Redundancy Checks

The simple additive checksums have a number of theoretical limitations, to do with errors they can and can't detect. The CRC methods have fewer problems.

```
srec_cat infile \
        --crop 0 0xFFFFFC \
        --fill 0x00 0x00000 0xFFFFFC \
        --b-e-crc32 0xFFFFFC \
    -o outfile
```

In the above example, we have an EPROM in the lowest megabyte of memory. The −crop filter ensures we are only summing the data within the EPROM, and not anywhere else. The −fill filter fills any holes left in the data. Finally, the −b-e-checksum-neg filter inserts a 32 bit (4 byte) checksum in big-endian format in the last 4 bytes of the EPROM image. Naturally, there is a little endian version of this filter as well.

The checksum is calculated using the industry standard 32-bit CRC. Because SRecord is open source, you can always read the source code to see how it works. There are many non-GPL version of this code available on the Internet, and suitable for embedding in proprietary firmware.

There is also a 16-bit CRC available.

```
srec_cat infile \
        --crop 0 0xFFFFFE \
        --fill 0x00 0x00000 0xFFFFFE \
        --b-e-crc16 0xFFFFFE \
    -o outfile
```

The checksum is calculated using the CCITT formula. Because SRecord is open source, you can always read the source code to see how it works. There are many non-GPL version of this code available on the Internet, and suitable for embedding in proprietary firmware.

You can look at the CRC of your data, by using the "hex-dump" output format.

```
srec_cat infile \
        --crop 0 0x10000 \
        --fill 0xFF 0x0000 0x10000 \
        --b-e-crc16 0x10000 \
        --crop 0x10000 0x10002 \
```

```
-o - --hex-dump
```

This command reads in the file, calculates the CRC of the data and places the CRC at 0x10000, crops the result to contain only the CRC, and then prints the checksum on the standard output in a classical hexadecimal dump format.

Where Am I?

There are several properties of you EPROM image that you may wish to insert into the data.

`srec_cat` *infile* `--b-e-minimum 0xFFFFFE 2 -o` *outfile*

The above example inserts the minimum address of the data (*low water*) into the data. This includes the minimum itself. If the data already contains bytes at the given address, you need to use an exclude filter. The value will be written with the most significant byte first. The number of bytes defaults to 4. There is also a little-endian variant, and two variants called "exclusive" that do not include the minimum itself.

`srec_cat` *infile* `--b-e-maximum 0xFFFFFE 2 -o` *outfile*

The above example inserts the maximum address of the data (*high water + 1*, just like address ranges) into the data. This includes the maximum itself. If the data already contains bytes at the given address, you need to use an exclude filter. The value will be written with the most significant byte first. The number of bytes defaults to 4. There is also a little-endian variant, and two variants called "exclusive" that do not include the maximum itself.

`srec_cat` *infile* `--b-e-length 0xFFFFFE 2 -o` *outfile*

The above example inserts the length of the data (*high water* + 1 - *low water*) into the data. This includes the length itself. If the data already contains bytes at the length location, you need to use an exclude filter. The value will be written with the most significant byte first. The number of bytes defaults to 4. There is also a little-endian variant, and two variants called "exclusive" that do not include the length itself.

What Format Is This?

You can obtain a variety of information about an EPROM load file by using the *srec_info*(1) command. For example:

```
$ srec_info example.srec
Format: Motorola S-Record
Header: "http://srecord.sourceforge.net/"
Execution Start Address: 00000000
Data:   0000 - 0122
        0456 - 0FFF
$
```

This example show that the file is a Motorola S-Record. The text in the file header is printed, along with the execution start address. The final section shows the address ranges containing data (the upper bound of each subrange is *in*clusive, rather than the *ex*clusive form used on the command line.

```
$ srec_info some-weird-file.hex --guess
Format: Signetics
Data:   0000 - 0122
        0456 - 0FFF
$
```

The above example guesses the EPROM load file format. It isn't infallible but it usually gets it right. You can use **−guess** anywhere you would give an explicit format, but it tends to be slower and not recommended.

MANGLING THE DATA

It is possible to change the values of the data bytes in several ways.

`srec_cat` *infile* `--and 0xF0 -o` *outfile*

The above example performs a bit-wise AND of the data bytes with the 0xF0 mask. The addresses of records are unchanged. I can't actually think of a use for this filter.

`srec_cat` *infile* `--or 0x0F -o` *outfile*

The above example performs a bit-wise OR of the data bytes with the 0x0F bits. The addresses of records are unchanged. I can't actually think of a use for this filter.

`srec_cat` *infile* `--xor 0xA5 -o` *outfile*

The above example performs a bit-wise exclusive OR of the data bytes with the 0xA5 bits. The addresses

of records are unchanged. You could use this to obfuscate the contents of your EPROM.

```
srec_cat infile --not -o outfile
```

The above example performs a bit-wise NOT of the data bytes. The addresses of records are unchanged. Security by obscurity?

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/