SPARKLE 3.2

USER MANUAL

Table of Contents

INTRODUCTION

Sparkle is an easy-to-use "all-in-one" cross-platform solution to creating multi-file Commodore 64 projects with an IRQ loader (demo, game, disk magazine, music collection, etc.), inspired by the loaders of Lft, Krill, and Bitbreaker. The Sparkle PC tool will generate D64 disk images using the information you provide in a text file called the loader script that will allow you to load your files, code, data, music, etc. in the order of your preference with a state-of-the-art IRQ loader and simple loader calls. You don't need to worry (or even know much) about file compression and decompression, loader installation, or disk layout, the Sparkle PC tool will take care of these and much more.

This manual will explain how you can use the Sparkle PC tool and the IRQ loader to create D64 disk images. For a description of new features please see VERSION HISTORY.

SUMMARY OF LOADER FEATURES

- Tested on 1541, 1541-II, 1571, and Oceanic drives, compatible with the 1541 Ultimate family.
- Resident size: `$2a0` bytes including loader, depacker, fallback IRQ (`$0160-$02ff`), and buffer (`$0300-$03ff`). The buffer contains preloaded data between loader calls, so it needs to be left untouched for sequential loading. The stack is reduced to `$0100-015f` but can be expanded if certain loader features are not used.
- Only three bytes are clobbered in the zeropage which can be selected from the script. Default is `$02-$04`. OK to use them between loader calls.
- 124-cycle GCR fetch-decode-verify loop tolerating disk rotation speeds of at least 272-314 rpm across all four disk zones providing high reliability. Checksum verification is done on-the-fly in disk zones 0-2 and partially outside the GCR loop in zone 3.
- C64 reset detection. The drive also resets if C64 reset is detected.
- 2Bit+ATN transfer protocol, 72 cycles/byte transfer speed. Transfer is freely interruptible.
- Spartan Stepping™ for seamless data transfer across adjacent tracks with zero additional stepper delay.
- Supports both sequential and bundle index-based loading with max. 128 directory entries and two alternative directory population methods.
- Built-in block-wise packer/depacker. The packer compresses file bundles back-to-back, eliminating zero-padding of the last sector of the bundles. This way no partially used sectors are left on the disk.
- Combined fixed-order and out-of-order loading.
- Bus lock. The loader uses `$dd00` for communication. By default, the user can freely alter `$dd02` between loader calls, as long as its value is restored before the next loader call. Bus lock via `$dd02` is also available if the user wishes to manipulate `$dd00`.
- Hi-score file saver plugin. Sparkle can save by overwriting pre-defined hi-score files. Saving is also freely interruptible.
- Custom drive code plugin. Upload your own drive code then restore Sparkle with simple calls.
- Loading to and saving from the RAM under the I/O registers (`$d000-$dfff`).
- 40-track disk support adding 85 sectors to the standard 35-track disk.
- PAL and NTSC support, including the plugins.

A SIMPLE PROJECT

Let's start with the simplest Sparkle project. Create a very simple PRG, something along the lines of

```
* = $2000

inc $d020
jmp *-3
```

Compile it and save it as `test.prg` in a `test` folder. Now open a text editor, create a new text file and add the following line:

```
File: "test.prg"
```

Save the file as `test.sls` in your `test` folder. For the sake of simplicity, also copy the Sparkle PC tool in the same directory. Then open a command line terminal, navigate to the `test` folder and run the following command on a Windows PC:

```
sparkle test.sls
```

or this one on Linux or macOS:

```
./sparkle test.sls
```

And that's it. You'll find a `test.d64` disk image in the `test` folder with a single entry in its directory. Loading and running this entry will install the loader which will then load and execute our simple test program.

In the following chapters we will discuss in detail how to create scripts, use the PC tool and the loader.

THE SPARKLE PC TOOL

While previous versions of the Sparkle PC tool had a GUI with a script editor and only worked on Windows machines, starting with version 3.0, it is now a cross-platform command-line tool. As discussed above, use the

```
sparkle script.sls
```

command format (or its Linux/macOS version) to create a D64 disk image. Sparkle only accepts loader script files with the `.sls` (Sparkle Loader Script) extension. The command-line argument should specify the path and file name of the script file. You can provide either the absolute path of the script or a path relative to the Sparkle PC tool.

On Windows, the Command Prompt window may close automatically when Sparkle exits (if it was not already open when Sparkle was started). To overcome this, you can use the optional command line parameter: `-p` with value `a` or `e`. Use this parameter if you want Sparkle to pause on exit and wait for `Enter` to be pressed before finishing. If you only want to pause on error, then use it with the value `e`. To always pause on exit, use it with the value `a`. The `-p` command line parameter must always be preceded by the script file name:

```
sparkle script.sls -p a
```

THE LOADER SCRIPT

The loader script is a simple text file consisting of a list of entries divided by line breaks and formatted into sections that are separated by empty lines. These sections provide all the necessary information required to create the desired disk image(s). Sparkle interprets entries sequentially from the beginning of the script file to its end. Here is an example script:

```
Path:       "path/name.d64"   << Path and file name of the D64 disk image
Header:     mydemoheader      << Directory header, max. 16 characters
ID:         dirid             << Directory ID, max. 5 characters
Name:       mydemoname        << First dir entry's name, max. 16 characters
Start:      xxxx              << Program's start address, 4-digit hex number
DirArt:     "path/file"       << File from which DirArt will be imported
Tracks:     35                << Number of tracks on disk, 35 vs. 40
IL0:        4                 << Interleave for tracks 1-17,  n = 1-14 (hex)
IL1:        3                 << Interleave for tracks 18-24, n = 1-12 (hex)
IL2:        3                 << Interleave for tracks 25-30, n = 1-11 (hex)
IL3:        3                 << Interleave for tracks 31+,   n = 1-10 (hex)
ProdID:     10aded            << Product ID, 6-digit hex number
ThisSide:   00                << This side's index, 2-digit hex number
NextSide:   01                << Next side's index, 2-digit hex number
ZP:         02                << Loader ZP address, set once per script

File:       "path/file0" aaaa bbbb cccc   << File #00 in Bundle #00

DirIndex:   01                << Directory index, 2-digit hex number (01-7f)
Align                         << Start this bundle in a new sector
Mem:        "path/file0" aaaa bbbb cccc   << Defines static memory segment
File:       "path/file1"* aaaa bbbb cccc << File #00 in Bundle #01 under I/O
File:       {path/file2} aaaa bbbb cccc   << File #01 in Bundle #01 unpacked

Script:     "path/script"     << Embedded script path and file name

Plugin:     common            << Include the Common Drive Code plugin

Plugin:     saver             << Include the Saver plugin

HSFile:     "path/hsfile" aaaa bbbb cccc  << Hi-score file
```

Script entries typically consist of an entry type identifier and one or more values separated by one or more TAB characters (except for the Align entry type). The number of TAB characters is arbitrary, and it is ok to use more than one to format the text. SPACE is only accepted as separator if the value of the entry is placed between "double quotation marks" or {curly brackets} (see FILE ENTRY SYNTAX). Entries not recognized by the Sparkle PC tool will be ignored, providing plenty of opportunity for commenting.

Entries can be categorized into three main groups: disk info, file bundle, and plugin entries. Disk info entries specify settings related to the disk image (things that you would define once per disk and project). Bundle entries define the files and data on the disk. Plugins allow uploading custom code to the

drive and file saving by overwriting predefined “hi-score” files. The presence of disk info entries will prompt Sparkle to finish any disk image in progress and start a new one. This way, it is enough to simply add a new disk info section after the last file bundle of the last disk to start the next disk in a multi-disk project, allowing you to create all the disks of the project using a single script. It is also possible, however, to use a separate script for each disk side in the project.

At minimum, the script must contain at least one `File` entry. Everything else is optional and can be omitted. If a disk info section does not precede the first bundle, then Sparkle will use default disk parameters for the first disk as outlined below (except if this script is meant to be embedded in another one in which case the disk section of the parent script will be used).

File paths in the script can be either absolute or relative. Relative paths are interpreted in relation to the script file’s absolute path on your computer.

DISK INFO

You can define all necessary settings related to the disk image here. The order of entries is arbitrary and any or all disk info script entries can be omitted in which case their default value is used, or the feature is not included on the disk. The following entry types are recognized:

`Path:` Here you can specify the final D64 file’s path and name. Sparkle will use this information when the disks are created. Both “\” and “/” are recognized and accepted as file path separators. If omitted the script’s path and name will be used to create a disk image.

`Header:` Max. 16 characters that will be used as the disk’s directory header which is on the left side of the topmost inverted row in the disk’s directory structure. You can omit this entry if you want to import it from a D64 or ASM DirArt file.

`ID:` Max. 5 characters that will be located on the right side of the header in the directory structure. You can omit this entry if you want to import it from a D64 or ASM DirArt file.

`Name:` The demo’s name. Max. 16 characters. This will be the first directory entry on the disk, and it will load the installer from track 18. You can omit this entry if you want to import it from a DirArt file. If a DirArt is not provided and the entry is omitted, then the script’s file name will be used as the first directory entry.

`Start:` The start address (program entry) of the demo. Once the loader is installed, it will load the first file bundle from the disk automatically and then it will jump to this address. If not specified, Sparkle will use the load address of the first file in the first bundle as a start address. (This is the only time the file order matters in a bundle.)

`DirArt:` There are 6 sectors available on track 18 for DirArt (max 48 directory entries, including the one specified at the `Name` script entry). Each DirArt directory entry will have a PRG file type and will start the demo but only the very first one will have a non-0 block size. Sparkle accepts several different file formats:

D64 files: Sparkle will simply import the directory structure of the D64 file. If the `Name` entry is omitted in the script, then the first DirArt entry’s type will be changed to PRG,

but otherwise Sparkle will keep the entry types in the DirArt. Sparkle will also import the header and the disk ID if the `Header` and `ID` entries are absent in the script.

TXT files: these are standard text files with line breaks after max. 16 characters. If a line is longer than 16 characters, then Sparkle will use the first 16 characters of it as a DirArt entry.

PRG files: Sparkle accepts two file formats: screen RAM grabs (40-byte long char rows of which the first 16 is used as directory entries, can be more than 25 char rows long) and $a0 byte terminated directory entries. If an $a0 byte is not detected sooner, then 16 bytes are imported per directory entry. Sparkle then skips up to 24 bytes or until an $a0 byte detected. Sparkle will ignore the first 2 address bytes of the PRG file. Example source code for $a0-terminated PRG (KickAss format, must be compiled):

```
* = $1000             //address can be anything, will be ignored
.text "hello world!"  //this will be upper case once compiled
.byte $a0             //terminates directory entry
.byte $30,$31,$32,$33,$34,$35,$36,$37,$38,$39,$01,$02,$03,$04,$05,$06,$a0
```

BIN files: Sparkle will treat this file type the same way as PRGs, less the address bytes.

ASM files: KickAss ASM DirArt source file. Please see Chapter 11 in the Kick Assembler Reference Manual for details. The ASM file may contain both disk and file parameters within [] brackets. Sparkle recognizes the 'filename' (=D64 path and name), 'name' (=directory header) and 'id' (=directory ID) disk parameters, and the 'name' and 'type' file parameters. All other parameters will be ignored. Disk parameters will only be imported if the `Path`, `Header` or `ID` entries are omitted in the script. If the Name entry is not provided in the script, then Sparkle will convert the first DirArt entry to PRG, otherwise it will import the entry types from the DirArt. Example:

```
.disk [name="dir header", id="-omg-"]
{
       [name = "0123456789ABCDEF", type = "prg"],
       [name = @"\117\$69\$75\$69\$B2\$69\$75\$69\$75\$ae\$B2\$75\$AE\$20
       \$20\$20", type="del"],
       [name = @"\$62\$62\$62\$62\$62\$62\$62\$62\$62\$62\$62\$62\$62\$A6
       \$A8\$A8", type="del"],
}
```

C files: Marq's PETSCII Editor C file. This file type can also be produced using Petmate. This is essentially a C source file which consists of a single unsigned char array declaration initialized with the directory entries. The first two values determine the border and background colors and are skipped, similar to the color RAM data that may follow the character codes. If present, Sparkle will use the 'META:' comment after the array to determine the dimensions of the DirArt. Sparkle will import max. 16 characters per screen row.

PET files: This format is supported by Marq's PETSCII Editor and Petmate. Sparkle will use the first two bytes of the input file to determine the dimensions of the DirArt, but it will ignore the next three bytes (border and background colors, charset) as well as color RAM data. Sparkle will import max. 16 characters per directory entry.

JSON files: This format can be created using Petmate. Sparkle will import max. 16 chars from each character row.

PNG files: Portable Network Graphics image file. Image input files can only use two colors (background and foreground). Sparkle will try to identify the colors by looking for a space character. If none found, it will use the darker of the two colors as background color. Image width must be a multiple of 128 pixels covering exactly 16 characters and the height must be a multiple of 8 pixels. Borders are not allowed. Image files must display the uppercase charset, and their appearance cannot rely on command characters. Sparkle uses the LodePNG library by Lode Vandevenne to decode PNG files.

BMP files: Bitmap image file. Same rules and limitations as with PNGs.

`Tracks:` Specifies whether this is a standard 35-track disk (default) or an extended, 40-track disk. This entry only accepts 35 or 40 as input. This entry is not needed for 35-track disks.

`IL0:` As well as `IL1:`, `IL2:`, and `IL3:`. These entries specify the sector interleave, i.e., the distance in sectors between two consecutive data blocks on the disk. The default values (if these entries are not used) are 4 for tracks 1-17 (`IL0`), and 3 for tracks 18-24 (`IL1`), 25-30 (`IL2`), and 31+ (`IL3`). Sparkle accepts decimal values between 1 and the number of sectors in a track less 1 (20, 18, 17, and 16, respectively) for each speed zone (the modulo of the number of sectors per track and the interleave in zone cannot be 0).

You can change the interleave to optimize Sparkle's performance in each speed zone separately, depending on how much raster time is left for loading. E.g., for tracks 1-17 (Interleave 0), use 4 if you have more than 75% of the raster time available on average for loading from the first 17 tracks, select an interleave of 5 if you have 50-75% of the raster time available, and an interleave of 7 may be appropriate if you only have about 25% of the screen time.

Please remember that the interleave can only be specified once for each speed zone on the disk and will be the same for each bundle within the speed zone. Finding the best interleave may need some experimentation. You may want to select one that works best for your most loading time sensitive part. Once the loading sequence enters a new speed zone on the disk, a new interleave can be used.

`ProdID:` Product ID. This is a unique identifier consisting of max. 6 hex digits which is used to identify disks that belong together in a multi-disk production. The Product ID is a global setting, it should be specified once and is shared by all the disks built from a single script. This way disks from another script will not be accepted during disk flipping. If a Product ID is not entered, then Sparkle will generate a pseudorandom number every time the script is processed. If you create disks of a single project from separate scripts, then enter the same Product ID and define `ThisSide` and `NextSide` IDs in each script (see below).

`ThisSide:` The current disk side's ID. A 2-digit hex number ($00-$7e) that identifies the current disk side. This is used when a new disk side is requested. By default, Sparkle assigns this automatically to subsequent disk sides, starting with $00. Use this entry along with the

same Product ID and a `NextSide` entry if you want to build disks of a project from separate scripts. It can be omitted if all disk sides are created from a single script.

`NextSide:` The next disk side's ID. A 2-digit hex number ($00-$7e) that identifies the next disk side in sequence. This is used when a new disk side is requested using a `LoadNext` call. By default, Sparkle assigns this automatically to subsequent disk sides, starting with $00. Use this entry along with the same Product ID and a `ThisSide` entry if you want to build disks of a project from separate scripts. It can be omitted if all disk sides are created from a single script. For further details see MULTI-SIDE PRODUCTIONS.

`ZP:` Zeropage usage, a 2-digit hex number. Sparkle uses 3 bytes in the zeropage. The default is `$02-$04`. If this suits your needs, the `ZP` entry can be skipped. If this interferes with your demo (e.g., SID uses the same addresses), then you can change it here. This is a global setting and should be specified once per script, in the first disk's info section, and it will be used for the rest of the demo. Use the same zeropage addresses if a separate script is used for each disk side. These zeropage addresses can be used freely between loader calls. Since loading typically runs from the main code, you can also save and restore them from IRQ during loading if needed.

FILE BUNDLES

Instead of loading files one by one, Sparkle can bundle them together and load them in batches. A file bundle is the sum of arbitrary files and data segments designated to be loaded during a single loader call. You can put any number of files and data segments in a bundle, provided they do not overlap in the memory. Files in a bundle don't need to occupy consecutive memory segments. The more you put in a bundle, the faster loading will be. A bundle may contain files for the next as well as a subsequent part. You can also combine files that are loaded in the RAM under the I/O area (`$d000-$dfff`) with others that are loaded to the I/O (e.g., color RAM) at the same memory address.

Each bundle consists of consecutive text lines forming a section in the script. The following entry type identifiers can be used in a bundle:

`DirIndex:` Sparkle uses its own internal directory for random access and by default assigns a directory entry index to each bundle, starting with $00, up to $7f. This, however, can be cumbersome in certain situations. E.g., a change in the order of the bundles in the script will also alter the bundles' directory entry indices. Also, many times only a handful of bundles require accessibility by index-based loading and the rest is only loaded using sequential loader calls. To resolve this issue, use the `DirIndex` entry type with a 2-digit hex number between $01-$7f to assign a specific directory index to a bundle. `DirIndex` value $00 is preserved for bundle 0 which for this reason doesn't require a `DirIndex` entry. If Sparkle finds at least one `DirIndex` entry in the script, then only bundles with a `DirIndex` will be added to the internal directory.

`Align` This is a parameterless bundle-specific setting (used without ":") that determines how Sparkle will add this bundle to the disk. By default, Sparkle saves compressed bundles back-to-back on the disk, leaving no unused space between bundles. This means that a

new bundle will first occupy the unused space in the last sector of the last bundle before starting a new sector. These transitional blocks are left in the loader buffer (`$0300–$03ff`) between loader calls and the next sequential loader call will first depack the beginning of the next bundle from the buffer before loading the next block. Add the Align keyword before the first file in the bundle to ensure Sparkle begins the new bundle in a new sector, leaving the last sector of the previous bundle partially unused. This can be helpful in loading time sensitive parts. However, most of the time, this option would just inflate disk usage.

`File:` The `File` entry is the essence of the script. `File` entries in a bundle must occupy consecutive text lines. An empty line in the script will trigger the start of a new bundle. When adding files to a bundle, you must provide the file's absolute or relative path and name followed by max. 3 file parameters that modify what is getting loaded from the file and where it is going to be loaded. By default, the first parameter is the load address (where the data will go). This is followed by the offset within the file (first byte of the file to be loaded). The last one is the length of the desired file segment (number of bytes to be loaded). Alternatively for files with a PRG header, if "-" (hyphen) is provided as the first parameter, then the second one will be interpreted as the memory address of the first byte to be loaded from the file, and the third parameter as the memory address of the last byte to be loaded. During disk building, Sparkle will only compress the selected segment of the file. For further details please see FILE ENTRY SYNTAX.

If a data segment overlaps the I/O area you can select whether it is to be loaded to the I/O area (e.g., VIC registers, color RAM, etc.) or in the RAM under the I/O area. If the file is destined in the RAM under the I/O area, mark the filename with an asterisk (*) which indicates that the loader will need to turn I/O off during loading of the specified data segment. Sparkle examines the I/O status of every data block separately and sets input/output register `$01` in the zeropage accordingly during decompression.

Sparkle sorts the files within a bundle during compression to achieve the best possible compression ratio. Therefore, file order within a bundle can be random in the script.

`Mem:` This entry defines a static memory segment that can be used as dictionary to improve the compression of the files in the bundle. If such a segment is provided in the script, Sparkle will rely on its content during loading and decompression of the files in the same bundle. Therefore, it is of utmost importance that the segment must be static and remain unchanged until loading and decompression of the bundle is complete. Examples of such segments include bitmaps, screen RAM, predefined tables, and code segments that are not self-modifying or are not in use during loading (e.g., code preloaded for the next part).

When Sparkle compresses the files in the bundle, it will also search the static memory segments described in the `Mem` entries in the same bundle for match sequences. The static memory segments can be anywhere in the memory, including under the I/O area, but not IN the I/O area which means that the color RAM, for example, cannot be used as a static memory segment (the reason for this is that the color RAM consists of nibbles,

and not full bytes so you cannot rely on the upper 4 bits). The most effective is, however, a memory segment that follows immediately the last byte of the file in the bundle. This is because Sparkle compresses and decompresses data "backwards" and by default match sequences are in in higher memory. Static memory segments in lower memory will be referred to with negative offsets that always require a 16-bit offset. On the other hand, match references in adjacent higher memory may only require an 8-bit offset.

The format of the `Mem` entry type is the same as that of the `File` entry type: first you must provide the path and name of a file that describes the static memory segment, then 0-3 file parameters.

FILE ENTRY SYNTAX

`File`, `HSFile`, and `Mem` entries must specify at least a file name with absolute or relative path and 0-3 file parameters.

- By default, the file name and file parameters are separated by an arbitrary number of tabulator characters.
- Alternatively, putting the file name in "double quotes" will allow the use of space as parameter separator.
- Using {curly brackets} instead of double quotes around the file name will instruct Sparkle to leave the file uncompressed and will also allow the use of space as parameter separator.

DEFAULT FILE PARAMETER SYNTAX

By default, the first file parameter after the file name is the load address of the file segment, the second one is an offset within the original file that marks the first byte to be loaded, and the last one is the length of the file segment. File parameters are hexadecimal numbers by default in word format (max. 4 digits) for the file address and file length, and double word format (max. 8 digits) in the case of the file offset. Hex prefix is not needed. Decimal number format can also be used by adding the "." prefix to the parameters. For example, in the following bundle the file entries will load 3 sections of a koala file to three different memory locations (note that the offset and file length parameters are in decimal format):

```
File:       path/koala.kla   4000  .0002      .8000
File:       path/koala.kla   6000  .8002      .1000
File:       path/koala.kla   d800  .9002      .1000
```

The first entry will load $1f40 bytes of bitmap data from offset 2 (i.e., skipping the first two address bytes) to memory address $4000. The second one $3e8 bytes of screen RAM data from decimal offset 8002 ($1f42) to memory address $6000. Finally, the third one will load $3e8 bytes from decimal offset 9002 ($232a) to the color RAM.

Parameters can be omitted but each one depends on the one on its left. I.e., you cannot enter the offset without first specifying the load address. In other words, the first file parameter will always be

interpreted as file address, the second one as offset, and the third one as length. Sparkle can handle SID and PRG files, so parameters are not needed for these file types unless you want to change them.

If all three parameters are omitted then Sparkle will load the file as a PRG file: it will use the file's first 2 bytes to calculate the load address, offset will be 2, and length will be (file length – 2). The only exception is files shorter than 3 bytes for which at least the load address is mandatory.

If only the load address is entered, then Sparkle will use 0 for the offset and the file's length as length.

If the load address and the offset are given but the length is not, Sparkle will calculate the length as (original file's length – offset), but max. (`$10000` - load address).

ALTERNATIVE FILE PARAMETER SYNTAX

Adding segments from PRGs to the disk and calculating offsets can be cumbersome. To simplify this, an alternative file parameter syntax can be used where only the address of the first and last bytes of the C64 memory segment to where we want to load must be specified (similar to how Kick Assembler displays the Memory Map with the addresses of different memory blocks). To use this syntax, the first file parameter must be "-" (hyphen), the second one is the memory address of the first byte, and the third one is the address of the last byte to be loaded. E.g. to load from Part.prg the file segment that goes to $2000-$2fff use the following syntax:

```
File: "Part.prg" - 2000 2fff     <<load from Part.prg to $2000-$2fff
```

This is equivalent to

```
File: "Part.prg" 2000 1801 1000
```

assuming that Part.prg loads to $0801. This syntax can be used with any other file extension (e.g. koala files) but keep in mind that Sparkle will assume that the first two bytes of the file are PRG header bytes (not part of the actual file) and will use them to calculate the load address.

FILE PARAMETER EXPRESSIONS

Sparkle also accepts mathematical expressions as file parameters in `File`, `HSFile`, and `Mem` entries. Parameter formulas must start with '='. When such a file parameter is found, Sparkle will evaluate the expression and calculate the file parameter. Please only use integers in expressions, fractions and constants are not supported. After a disk side is completed, Sparkle will list the evaluated expressions and their results in the output window. This allows you to double check your formulas to make sure the file parameters are calculated correctly.

Expressions must use the same number formats as described in the FILE ENTRY SYNTAX chapter. In short, any number without a prefix will be interpreted as a hexadecimal number. For decimal numbers, use the "." prefix. For example, after evaluation of the expressions in following entry

```
File:      path/myfile.prg  =2000+(4*.256)   = 2 + (4 * .256)      100
```

$2400 will be used for load address, and $0402 for offset. Technically, any file parameter could be entered as an expression (e.g., =2000), but this would just result in overcrowding of the output window making it difficult to hunt down expression errors.

Sparkle uses the TinyExpr++ library by Blake Madden (which is based on the TinyExpr library by Lewis Van Winkle) to evaluate math formulas.

FILES UNDER I/O

By default, Sparkle writes `#$35` to input/output register `$01` and turns the BASIC and KERNAL ROMs off during loading but leaves I/O on. If any part of a file is to be loaded in the RAM under the I/O area (`$d000-$dfff`), an asterisk (*) must be added to the end of the file name (see template above). This will instruct Sparkle to turn off the I/O area while unpacking the file. If the * is omitted the file will be loaded to the I/O area (VIC, SID, and CIA registers, color RAM etc.). A bundle may contain two file segments sharing the same load address if one is destined under I/O and the other one is to be loaded to the I/O area.

EMBEDDED SCRIPT ENTRIES

Scripts can become very long, so you may want to create shorter scripts, and then embed these in your main script. This is done by adding the `Script:` entry type as a separate "bundle" followed by the embedded script's path and file name. When Sparkle reaches a `Script` entry during disk building, it will first process its content before continuing with the next entry. You can add whole disks to your script this way. If `File` entries in an embedded script use relative file paths, then the embedded script's absolute path will be used to calculate the files' absolute path.

Scripts cannot be inserted into an existing file bundle. Files in the embedded script will always start a new file bundle and won't be added to the current bundle. In other words: you can't have both `File` and `Script` entries in the same bundle. The `DirIndex` and `Align` entry types are allowed in bundles with `Script` entries, but it may be a better idea to include these in the embedded script.

PLUGINS AND HI-SCORE FILES

Plugins and hi-score files can be placed anywhere in the loader script after the disk info section, but it is recommended to group them either before or after the file bundles. Each plugin and hi-score file entry must be in a separate bundle. On the disk, plugin and hi-score files will follow the last regular file bundle. This way, all regular file bundle will remain accessible via sequential loading. In contrast, plugins and hi-score file can only be accessed by index-based loading. You can use their bundle index or assign a `DirIndex` entry type to `Plugin` and `HSFile` entries. However, this way only bundles with a `DirIndex` can be loaded using indexed loading calls. To overcome this problem, the `PlgIndex` entry type is introduced. The following script entries are used to define plugins and hi-score files:

`Plugin:` Specifies a loader plugin. Currently `saver` and `custom` are accepted as values, for the file saver and the custom drive code plugins, respectively. Each plugin entry will use 2 sectors on the disk.

`HSFile:` Hi-score file. This entry adds a file to the disk that can be later overwritten using the saver plugin. The syntax of the `HSFile` entry is the same as that of the `File` and `Mem` entries. You can add multiple hi-score file entries to the disk, but each `HSFile` entry must be preceded by its own saver plugin entry. You can also create blank hi-score files by using `blank` as filename, and providing a load address, `0000` as offset, and a file length. For blank hi-score files all 3 file parameters are mandatory.

`PlgIndex:` Plugin-specific directory index. Like the `DirIndex`, the `PlgIndex` entry type assigns a directory index to the plugin entries. Using `PlgIndex` entries, however, will not restrict index-based loading and all bundles will remain accessible using their bundle indices. Therefore, `PlgIndex` values must be greater than the number of file bundles on the disk (and cannot exceed the capacity of the internal directory). It is recommended that a `PlgIndex` is assigned to every plugin bundle. `PlgIndex` entries can also be combined with `DirIndex` entries.

COMMENTING

Sparkle allows commenting the script and in general, it will interpret any unrecognized entries as comments. Comments can be added essentially anywhere, provided they are in a separate text line or follow an entry in the same line but are separated by one or more tabs from the entry:

```
<< Block comment
File:      path/file  xxxx  yyyyyyyy   zzzz       << inline comment
```

In the above example the '<<' sign is only a visual aid. It is not used by Sparkle to determine whether an entry is a comment or not. You can choose the comment identifier of your preference or use none as long as the comment cannot be misinterpreted as a standard entry.

MULTI-SIDE PRODUCTIONS

To create a multi-disk project simply add the second disk's info section to your script after the last bundle of the first disk (separated by at least one empty line) then start adding its file bundles. For disk flipping to work, Sparkle must be able to determine whether the inserted disk is part of the production and if it is the requested disk side. I.e., side B of demo B should not be accepted after side A of demo A. Sparkle uses the `ProdID`, `ThisSide`, and `NextSide` entries for this purpose. The `ProdID` is ideally a unique 6-digit hex number shared by all disk sides in the project. The `ThisSide` index is unique to the disk side, and the `NextSide` index determines which disk side should follow the current one in sequence (must be equal to the `ThisSide` index of the next disk side).

The simplest way to create a multi-side production is building all disk sides from a single script to make sure disks are properly numbered and use the same Product ID. This way, the `ProdID` only needs to be defined once, in the first disk's info section, and the `ThisSide` and `NextSide` entries can be omitted. You may create a separate script for each disk side and then embed them in the main script using the `Script` entry type. Disks are automatically assigned a disk side index in creation order, starting with

$00 for the first disk. You can assign a `NextSide` index to the last disk if you want your production to loop.

The other option is building each disk side from a separate script. In this case, you must enter the same `ProdID` and must specify a `ThisSide` and a `NextSide` entry in each script. Typically, the `ThisSide` index of the first disk side is $00, and the `NextSide` index will be $01 which will be the `ThisSide` index of the second disk, etc., but any numeric sequence can be used as long as all indices are between $00-$7e ($7f is reserved for resetting the drive, see LOADER FUNCTIONS). In general, the `NextSide` index of a disk side and the `ThisSide` index of the subsequent disk side must always be the same. The `NextSide` index of the last disk side must be omitted, except in the case of a looping production. If the `Path` entry with the D64 file's name is not specified, then Sparkle will use the `Name` entry and add _SideA/B etc. to it to create a file name for each disk side.

For details about disk flipping, please see REQUESTING A DISK SIDE.

RUNTIME CONSIDERATIONS

In the previous chapters we discussed how to create a loader script that the Sparkle PC tool can use to build a D64 disk image. This chapter is about how you can use the IRQ loader to access your files on a Sparkle disk.

Sparkle's own code is stored on track 18. The installer and the C64 resident code take 4 blocks, the drive code an additional 6 blocks, and the internal directory another 2 blocks. The remaining 6 blocks on track 18 can be used for DirArt. This leaves the entire 664 blocks of a standard 35-track disk for your project.

All directory entries in a Sparkle disk's standard directory point at the installer but only the first entry shows the block count. Loading any of the directory entries will load the installer and not the first file or bundle on the disk. When you run the installer, it will install the C64 resident code and the drive code, set the I flag, write `#$35` to `$01`, and reduce the stack to the lower `$60` bytes. After this, it will automatically load the first file bundle.

Once the first bundle is loaded, the loader will restore input/output register `$01` to `#$35` and it will jump to the start address as specified in the script (or to the first byte of the first file in the script if a start address was not entered) without clearing the I flag (IRQs remain disabled). The installer and the loader do not alter any other vectors or VIC registers.

During loader calls Sparkle loads each block from the disk into the loader buffer, then decompresses it to its destination from there before the next block is loaded. The loader writes `#$35` to `$01` at the beginning of every loader call and will return to your code with this value in `$01`. During decompression, the value of `$01` can be either `#$34` or `#$35` depending on the destination of the files in the bundle. Loader calls do not clobber the I flag.

Once loading of a bundle is finished, the loader buffer will contain the first sequence of the next bundle. This is the result of Sparkle's compression scheme which fills up each block on the disk completely. Thus, the next bundle starts in the same block, right after the previous one, leaving no space unused on the disk. The next sequential loader call will first decompress the already loaded initial sequence of the next

bundle from the buffer before the next block gets transferred from the drive. Therefore, it is imperative to leave the loader buffer untouched between sequential loader calls. Indexed loader calls, on the other hand, always reload the buffer. So, if you need to overwrite the loader buffer between loader calls, you must make sure that the next bundle is loaded using an indexed loader call.

The C64 resident code resides at $0160-$02ff and the loader buffer is at $0300-$03ff. The C64 resident code has the following memory layout:

```
      $0160-$0179     Sparkle_IRQ
      $017a-$019b     Sparkle_SendCmd
      $019c-$019e     Sparkle_LoadA
      $019f-$02ff     Sparkle_LoadNext + depacker
      $0300-$03ff     Loader buffer
```

This layout allows freeing up more stack space if certain functions are not required in your project. E.g., if you want to use your own fallback IRQ then you can set the stack pointer to $79. If your project also only uses sequential loader calls, then you can further extend the stack to $9e.

Alternatively, you can create macros for the Sparkle_SendCmd and Sparkle_LoadA calls if they are only used very infrequently to save on the stack space. Here are two macros in KickAss format to replace the Sparkle_SendCmd and Sparkle_LoadA functions:

```
.macro Sparkle_RequestBundle(DirIndex)//same as Sparkle_SendCmd
{
      .var Bits       = $04           //default ZP address
      .var sendbyte   = $18
      .var buslock    = $f8

      lda #DirIndex
      sta Bits
      lda #$35
      sta $01
      ldx #sendbyte
      stx $dd00
      bit $dd00
      bmi *-3
!:    adc #$e7
      sax $dd00
      and #$10
      eor #$10
      ror Bits
      bne !-

      lda #buslock
      sta $dd00
}
```

```
.macro Sparkle_LoadBundle(DirIndex)    //same as Sparkle_LoadA
{
      Sparkle_RequestBundle(DirIndex)
      jsr Sparkle_LoadFetched
}
```

LOADER FUNCTIONS

The `resources` folder in Sparkle's source code includes a Sparkle.inc file with all the important loader addresses in Kick Assembler format. This file can be recreated by assembling the SL.asm source file in the same folder. Please use the `-afo` KickAss switch, otherwise the generation of this file may be blocked. Note that assembling the C64 source files is not required for using Sparkle.

From your code, the following functions are available:

- Sparkle_LoadNext (`JSR $21c`)

  Sequential loader call, parameterless, loads next bundle in sequence

  This parameterless call will load the next bundle of files in sequence, as specified in the script. When the loader is called, it first depacks the first partial block of the next bundle from the buffer before receiving the next block from the disk. Therefore, the loader buffer must be left untouched between sequential loader calls. The I/O area may be turned on or off during depacking depending on where the data is designated.

  Once the very last bundle is loaded from a disk, the next sequential loader call will instruct Sparkle to check whether a subsequent disk side is expected. In case of a multi-disk demo, the loader will move the read/write head to track 18 and wait for the new disk side to be inserted. Once the new disk side is detected the first file bundle is loaded automatically. If there are no more disks this loader call will reset the drive. Do not use this function to reset the drive as the loader call may enter an endless loop.

- Sparkle_LoadA (`JSR $19c`)

  `A=$00-$7f` – bundle index-based loader call, or
  `A=$80-$fe` – requests a new disk & loads first bundle automatically

  This call will load the bundle specified by the index in A. Sparkle can handle bundle indices between `$00-$7f` (0-based). Sectors 17-18 on track 18 are used as internal directory. One directory entry requires 4 bytes, so 64 bundle indices can be accessed per directory block. There is space for only one of the two internal directory blocks in the drive's RAM at a time. If the bundle index is outside the range of the directory block in the drive's RAM, the other block will be fetched from track 18 before the requested bundle gets loaded. See more about Sparkle's internal directory structure here.

  To request another disk side, use `$80-$fe` ($80 + disk index) in A (`$ff` is reserved for drive reset). Do not use this function to reset the drive (see Sparkle_SendCmd).

```
lda #bundle_index          //loads bundle with index in A
jsr Sparkle_LoadA

lda #$80+disk_index        //requests disk with index in A
jsr Sparkle_LoadA          //then auto-loads first bundle
```

- Sparkle_SendCmd (`JSR $17a`)

  `A=$00-$7f` – requests a bundle and prefetches its first sector, or
  `A=$80-$fe` – requests a new disk without auto-loading its first bundle, or
  `A=$ff` – resets the drive

  This function can be used to send commands to the drive without necessarily loading anything. Positive values in A will instruct the drive to find the first sector of the requested bundle. The drive will fetch this sector without transferring it to the C64. Thus, the read/write head can be positioned, and the drive readied ahead of loading a bundle. Use it in combination with Sparkle_LoadFetched:

```
lda #bundle_index          //prefetches first sector
jsr Sparkle_SendCmd        //of requested bundle
…
jsr Sparkle_LoadFetched    //loads prefetched bundle
```

  You can also use this function to request a disk by its disk index, without auto-loading the first bundle from the new disk:

```
lda #$80+disk_index        //requests new disk
jsr Sparkle_SendCmd        //without loading first bundle
…
lda #bundle_index          //loads requested bundle
jsr Sparkle_LoadA          //once new disk is detected
```

  Finally, calling Sparkle_SendCmd with `A=$ff` will reset the drive.

- Sparkle_LoadFetched (`JSR $19f`)

  Loads a prefetched bundle, use only after fetching a bundle with Sparkle_SendCmd (A=bundle_index). See example above.

OTHER IMPORTANT ADDRESSES

- Sparkle_IRQ (`$0160`)

  This is a very simple fallback IRQ which contains a single JSR instruction (Sparkle_IRQ_JSR, see below). You can use this IRQ between parts for example. It also allows loading under the I/O area. If it is not used, then the stack pointer can be set to $7a.

- Sparkle_IRQ_JSR (`$016e`)
  This is the address of a JSR instruction within the fallback IRQ that at start points at an RTS instruction. You can change it to music player call, for example, if you want to use the fallback IRQ while loading the next part.

- Sparkle_IRQ_RTI (`$0179`)
  This is the address of the RTI instruction at the end of the fallback IRQ.

THE INTERNAL DIRECTORY

Sparkle uses its internal directory for random bundle access. The internal directory takes the last two sectors (512 bytes) of track 18. Each directory entry uses 4 bytes; thus, there can be maximum 128 directory entries. Entries have the following format:

```
$00   Track
$01   First sector on track in sequence (not first sector of bundle)
$02   Sectors remaining on track
$03   Offset of first byte in first sector of bundle
```

The loader uses the first three bytes of the entry to correctly identify the sector that contains the beginning of the requested bundle. The first one is the track where the bundle starts. The second byte marks the first sector on the track when the track is being read in sequence determined by the interleave. Sparkle doesn't start each track with sector 0. To determine the first sector in sequence on the next track, it simply adds the interleave to the last sector of the previous track. The third directory entry byte is the number of sectors remaining on the track if we start reading from the requested bundle's start sector. Sparkle uses this first to mark off all sectors on the track that preceded the requested bundle then to identify the sectors that belong to the requested bundle. The fourth byte is an offset within the sector to the first byte of the bundle.

In the drive's memory there is one page allocated for the internal directory. This means that we can have 64 entries in the RAM. By default, the first block of the internal directory is loaded. If an entry is requested that is not currently in the drive's RAM, then Sparkle will first need to fetch its directory block before the requested bundle can be loaded.

Sparkle can handle two different directory entry assignments:

- The default is when no `DirIndex` entries are used in the script. In this case all bundles will be added to the internal directory, starting with bundle 0 and up to bundle 127. Thus, bundles can be loaded using index-based (`Sparkle_LoadA`) loader calls with their bundle indices. The disk can have more than 128 bundles but only the first 128 can be accessed using bundle index-based loading. The downside of this approach is that bundle indices will change if the bundle order changes on the disk. Sparkle doesn't keep track of the last loaded bundle's index, so the user will need to handle this from their code.

- The other option is assigning a `DirIndex` to the bundles that will be accessed using indexed loading. In this case, only bundles with an assigned `DirIndex` will be added to the directory and the rest will be only accessible using sequential (`Sparkle_LoadNext`) loader calls. The

`DirIndex` can be any (random) value between $01-$7f. `DirIndex` $00 is preserved for bundle 0 and is used internally. This way, a constant directory index can be used, even if the bundle order changes in the script. While the `DirIndex` can be random, using values $01-$3f (if less than 64 bundles will require index-based loading) will allow accessing all these bundles in one internal directory block. Thus, we won't need to reload the directory.

LOADING TO THE RAM UNDER I/O REGISTERS ($D000-$DFFF)

Sparkle will change the value of input/output register $01 to #$34 when loaded data gets decompressed to the RAM under the I/O registers. Thus, I/O registers (VIC, SID, etc.) may not be directly accessible when loading is interrupted by an IRQ. Therefore, if you want to load under I/O, you must also make sure your code can handle this situation. Specifically, IRQs must save the status of $01 at start then set it to #$35 and finally, restore it. Here is the (slightly modified) source of Sparkle_IRQ to demonstrate this:

```
Sparkle_IRQ:     pha
                 txa
                 pha
                 tya
                 pha
                 lda $01
                 pha              //save $01 to stack
                 lda #$35         //set $01 to #$35 to allow access
                 sta $01          //to I/O registers
                 inc $d019
Sparkle_IRQ_JSR: jsr MusicPlay
                 pla
                 sta $01          //restore $01 from stack
                 pla
                 tay
                 pla
                 tax
                 pla
Sparkle_IRQ_RTI: rti
```

SWITCHING VIC BANKS

Sending data from the C64 to the floppy drive relies on the state of two registers in the C64. Bits 3-5 in the CIA 2 Data Direction Register A ($dd02) turn data transmission from the C64 to the drive on or off, and the corresponding bits in the CIA2 Port A ($dd00) set the actual value of the transmitted data. The C64 resident code of Sparkle uses $dd00 to communicate with the drive. The same register is also often used in demo effects to switch VIC banks. Changing the value of $dd00 by the user, however, may be misinterpreted by the loader as a drive command and could result in unwanted loader behavior such as premature reset of the drive. To avoid this, you must use $dd02 to select a VIC bank, using the following formula:

```
lda #$3c + VIC_Bank
sta $dd02
```

where the value of `VIC_Bank` is 0 for VIC bank 0, 1 for bank 1, etc.

Some demo effects need to set the VIC bank and other registers using the same value requiring values different from the above formula in the register. To allow this, Sparkle implements a bus lock.

BUS LOCK

Bus lock means setting either $dd00 or $dd02 to a specific value that allows free manipulation of the other register without transmitting data to the floppy drive.

Imagine the following scenario. You have a faucet with two handles: Handle A turns water flow on and off, and Handle B sets water temperature from cold ("off") to warm ("on"). Let's start with warm water flow turned on (both handles are "on"). Our goal is to be able to freely manipulate one of the two handles *without releasing cold water*. We can achieve this in two different ways:

- We can turn water flow off using Handle A which then allows us to select any state of Handle B, and we can be sure that there won't be any cold water released.
- We can keep Handle B on "warm" and then freely manipulate Handle A (i.e., turn water flow off and on) to achieve the same goal. Either way, the output of the faucet will never be cold water.

In the above "faucet lock" scenario, Handle A corresponds to $dd02, and Handle B to $dd00. When the loader is installed, bits 3-5 (ATN out, Clock out, Data out) on $dd02 are set to 1 allowing data to be transmitted to the drive. Then, when a loader job is complete, the loader sets the corresponding bits on $dd00 to 1 before the loader call returns to your code. From this state, you can do two different things:

1. As long as all 3 bits on $dd00 are left unchanged, you can freely manipulate all bits on $dd02 (all possible values from $00 to $ff are permitted) – the drive won't notice anything. We may call this "indirect" bus lock as we don't necessarily prevent data from reaching the drive (depending on the set value of $dd02), we just make sure that if data is received it always remains the same (as the value of $dd00 doesn't change).

2. The other option is first clearing bits 3-5 on $dd02, thereby preventing any data from being transmitted from the C64 to the drive. After this, you can set $dd00 to any value ($00-$ff) without affecting the drive. We may call this approach "direct" bus lock as we turn off data transmission to the drive.

<u>In both cases, the original state of both $dd00 and $dd02 must be restored before loading operations can resume.</u>

INDIRECT BUS LOCK TO ALLOW UNRESTRICTED USE OF $DD02

Once a loader job is complete, indirect bus lock is turned on by default and the loader is natively ready for any manipulation of $dd02, as long as the #$3c + VIC_Bank format is restored before the next loader call. Here is an example code snippet:

```
      jsr Sparkle_LoadNext
                           //indirect bus lock is on by default
      ldx #$00
Loop:
      stx $dd02            //any value permitted on $dd02
      inx                  //indirect bus lock remains active
      bne Loop

      lda #$3c+VIC_Bank    //#$3c-#$3f for VIC banks 0-3
      sta $dd02            //restore default value on $dd02…

      jsr Sparkle_LoadNext //…before next loader call
```

DIRECT BUS LOCK TO ALLOW FREE MANIPULATION OF $DD00

We need some extra preparations for direct bus lock. The goal is to make sure either direct or indirect bus lock is always on. Indirect bus lock is active by default. We start by turning direct bus lock on, before we can turn indirect bus lock off by overwriting $dd00:

```
      jsr Sparkle_LoadNext
                           //indirect bus lock active by default
      lda #$03
      sta $dd02            //turn on direct bus lock
                           //both direct and indirect bus locks are on
      ldx #$00
Loop:
      stx $dd00            //any value permitted on $dd00
      inx                  //direct bus lock remains active
      bne Loop


      lda #$38             //restore indirect bus lock
      sta $dd00            //direct bus lock is still active
      lda #$3c+VIC_Bank    //restore default value on $dd02…
      sta $dd02            //…turning direct bus lock off

      jsr Sparkle_LoadNext
```

Step-by-step instructions for unrestricted use of $dd00:

1. Turn direct bus lock on by writing #$03 to $dd02. Technically, any value with bits 3-5 cleared works. Setting bits 0-1 to 1 allows the conventional (inverted) VIC bank selection on $dd00.
2. Write anything to $dd00 as required by your program.
3. Restore $dd00 to #$38 to activate indirect bus lock. (Sparkle uses the value #$f8 to the same effect due to memory constraints.)
4. Restore $dd02 to #$3c+VIC bank (#$3c-#$3f) - this will also deactivate direct bus lock.
5. Step 3 must ALWAYS precede step 4 to make sure either direct or indirect bus lock always remains active.

REQUESTING A DISK SIDE

Sparkle stores the current and the next disk's indices in the BAM. From your code, there are three different ways to request the next or any other disk side.

1. Once the final bundle of a disk side is loaded, the next Sparkle_LoadNext sequential loader call will move the read/write head to track 18 and Sparkle will keep fetching the BAM until the next disk's index is detected. After this, it will automatically load the first file bundle from the freshly inserted disk.
2. You may use the Sparkle_LoadA call with `A=#$80+disk_index` to request an arbitrary disk side. Sparkle will keep fetching the BAM until the requested disk index is detected. Once the requested disk is inserted, Sparkle will automatically load the first file bundle from the new disk.
3. To skip loading the first bundle after disk change, use the Sparkle_SendCmd function with `A=#$80+disk_index`. Once the requested disk side is inserted, Sparkle will fetch the first block on the disk but will not transfer it. After this you can either use the Sparkle_LoadFetched function to load the first bundle (which is essentially the same as option 2) or the Sparkle_LoadA function with a bundle index in A to load any other file bundles.

DISK FLIPPING USING NONBLOCKING CALLS

Sparkle's loader calls (Sparkle_LoadA, Sparkle_LoadNext, Sparkle_LoadFetched) are blocking calls which means that a disk flip call will not return to your program until the new disk side has been detected and the first block of the first bundle has been loaded. This can limit what you can do in a turn-disk part to using IRQs only. The following two code snippets will allow you to overcome this obstacle.

REQUESTING A DISK SIDE AND LOADING THE FIRST BUNDLE OF THE NEW DISK

1. Let's start by requesting the next disk side using the Sparkle_SendCmd function in the main loop. This call will return immediately after the byte was sent to the drive.

```
lda #$80+disk_index
jsr Sparkle_SendCmd
```

2. The drive is now waiting for the requested disk side to be inserted. Next, still in the main loop, we let the drive know that we will want to load the first, automatically fetched bundle by sending a "ready-to-receive" signal.

```
lda #$08
sta $dd00
```

3. Now we need to know when the drive has detected the next side and is ready to send the first block of the first bundle. To achieve this, we will poll $dd00. This can be done in either the main loop or in an IRQ. If the N flag is clear after polling, then the next disk side has been detected, and the drive is ready to transfer the first block. If the N flag is set, then the drive is not ready yet. In this case we can continue with our disk flip part and keep polling the drive periodically.

```
PollDrive:                  //main loop example
     bit $dd00
     bpl DriveReady         //N=0, drive is ready to send
     ...                    //N=1, drive is not ready, continue part
```

```
      jmp PollDrive
```

4. Once the drive is ready, we can load the first bundle in the main loop.

```
DriveReady:
      jsr Sparkle_LoadFetched
```

REQUESTING A DISK SIDE AND LOADING ANY BUNDLE FROM THE NEW DISK

1. We start again by requesting the next disk side from the main loop.

```
      lda #$80+disk_index
      jsr Sparkle_SendCmd
```

2. Next, we need to let the driver know that we want to send a bundle index by sending a "sendbyte" signal.

```
      lda #$18
      sta $dd00
```

3. Then we wait for the drive to be ready to receive the bundle index after the new disk side was inserted by periodically polling $dd00 and watching the N flag.

```
PollDrive:                  //main loop example
      bit $dd00
      bpl DriveReady        //N=0, drive is ready to receive
      ...                   //N=1, drive is not ready, continue part
      jmp PollDrive
```

4. Finally, we load our wanted bundle when the drive is ready using an index-based loader call.

```
DriveReady:
      lda #bundle_index
      jsr Sparkle_LoadA
```

LOADER PLUGINS

Loader plugins extend the functionality of Sparkle beyond loading. They consist of drive-side code loaded to the stack in the drive's RAM and C64-side routines loaded to the loader buffer at $0300-$03ff on the C64. They are not included on the disk by default, so additional steps must be taken to access these functions. Currently, two plugin types are available: the hi-score file saver plugin and the custom drive code plugin. Loader plugins and hi-score files can be only accessed by index-based loader calls.

Although it is not a requirement, the use of `DirIndex` entries is recommended when plugins are included on the disk.

HI-SCORE SAVER PLUGIN

Sparkle can overwrite predefined "hi-score files" on the disk. To include this plugin on the disk, add the following lines to your script, after the disk info section, before or after the last file bundle:

```
PlgIndex:  7e          <<plugin directory index
Plugin:    saver

PlgIndex:  7f
HSFile:    "path/hsfile" aaaa bbbb cccc
```

The `Plugin` and the `HSFile` entries are separate bundle and must be separated by an empty line. Sparkle will use the load address of the hi-score file during file saving. The size of a hi-score file must be rounded to the nearest $100 bytes. The saver code requires 2 sectors, and the raw, uncompressed hi-score file will use 2+ sectors. Each hi-score file entry must be preceded by its own saver plugin, but otherwise, the number of saver plugin + hi-score file entry combos is only limited by the number of available directory slots. The use of `PlgIndex` directory index entries is recommended but not mandatory.

To activate the file saving feature, one must first load the saver plugin code by issuing a Sparkle_LoadA call using the saver plugin's directory index. The drive will first fetch and transfer the first of the 2 saver plugin blocks containing the C64 side of the saver code. Then it will fetch the second block of the saver code in the drive's buffer and will execute it there entering a loop waiting for data transfer from the C64. Once the plugin is loaded, Sparkle is ready to overwrite the hi-score file that directly follows the plugin on the disk. The second call in the snippet below (Sparkle_Save) will save the number of pages specified in A. The size of the file must be a multiple of `$100` bytes. Sparkle will save from the load address of the original hi-score file, as defined in the script. The new file can occupy less but not more sectors on the disk than the original hi-score file. Calling the saver code with `A=#$00` or with a value greater than the high byte of the original hi-score file's size will instruct the loader to return to normal loading operations without saving anything. Once saving is complete, Sparkle's drive code exits the saver loop and returns to normal loading operations. The next loader call must be index-based as we are at the end of the disk, after standard file bundles. To re-save the hi-score file, the saver plugin must be reloaded.

Sparkle_Save (`JSR $302` – A = high byte of file size, #$00 aborts without saving)

```
lda #PlgIndex          //Plugin directory index of saver plugin
jsr Sparkle_LoadA      //load the saver plugin to the buffer
lda #>FileSize         //A=#$01+ to save, A=#$00 to abort saving
jsr Sparkle_Save       //overwrite hi-score file
…
lda #PlgIndex          //Plugin directory index of hi-score file
jsr Sparkle_LoadA      //reload hi-score file (if needed)
```

CUSTOM DRIVE CODE PLUGIN

This plugin consists of the drive-side and C64-side routines necessary to upload and execute custom code in the drive's RAM and then restore Sparkle's drive code once the custom drive code is no longer used. Normal loading operations are not available while custom code is running in the drive's memory. To include the plugin on the disk the following entry must be added to the script after the last file bundle:

```
PlgIndex:   7f
Plugin:     custom
```

Like the saver plugin, the custom drive code plugin will occupy 2 sectors on the disk, and it must be loaded with a Sparkle_LoadA call using the plugin's bundle/directory index.

During standard operation, Sparkle's drive code uses the zeropage and $0300-$07ff in the drive's RAM. Pages 1 and 2 are used as block buffers. The plugin's drive memory resident portion uses $0100-$017f. Therefore, the whole zeropage, $0180-$01ff in the stack, and $0200-$07ff may be utilized for custom drive code, and code can be directly uploaded to the zeropage and to $0200-$07ff. The user may choose to use part or the entire available drive memory, but the order by which the memory pages can be accessed is predetermined (see below).

The use of the plugin requires the following steps.

1. Transferring and storing some or all of Sparkle's drive code (0-6 pages) on the C64.

   Sparkle_RcvDrvCode (`JSR $0302`)

   A = number of pages to be transferred from the drive to the C64 (#$00-#$06)
   X = C64 destination address high byte

   This function call is mandatory, even if nothing needs to be transferred. Depending on the custom drive code's memory requirements, one can chose to transfer and store 0-6 blocks of Sparkle's drive code. $0200-$02ff and $0180-$01ff in the stack are always available for custom code. The order by which blocks are transferred is not freely selectable and is determined by the value of A. Note that the zeropage gets always transferred last when more than one block is transferred.

   A = 0 Nothing is transferred. 0200-$02ff will be available for custom code upload. Alternatively, the entire drive RAM can be used if we don't want to restore Sparkle.
   A = 1 The zeropage is transferred and saved at the memory address specified in X. $0200-$02ff and the zeropage will be available for custom code upload.
   A = 2-6 1-5 pages from $0300 upwards and the zeropage are transferred and saved back-to-back at the memory address specified in X. $0200-$02ff plus 1-5 pages from $0300, and the zeropage will be available for custom code upload.

Value of A	$0100-0$1ff	$0200-$02ff	$0300-$03ff	$0400-$04ff	$0500-$05ff	$0600-$06ff	$0700$07ff	ZP
0	---	---	---	---	---	---	---	---
1	---	---	---	---	---	---	---	B1
2	---	---	B1	---	---	---	---	B2
3	---	---	B1	B2	---	---	---	B3
4	---	---	B1	B2	B3	---	---	B4
5	---	---	B1	B2	B3	B4	---	B5
6	---	---	B1	B2	B3	B4	B5	B6

The order in which drive memory pages are transferred is determined by the value of A
(--- = not transferred; B1 – B6 = block order of transfer)

2. Uploading (and executing) the custom drive code to the drive (0-7 blocks)

Sparkle_SendDrvCode (`JSR $0380`)

A = number of custom drive code pages (#$00-#$07)
X = C64 source address high byte

Once we create space on the drive, the next step is uploading the custom drive code. The plugin's drive memory resident part uses $0100-$017f, so custom code may occupy $0200-$7ff plus the zeropage in the drive's RAM and may also use $0180-$01ff in the stack. As in the previous step, the order by which custom code blocks are uploaded is not freely selectable and is determined by the value of A.

A = 0 No upload, abort
A = 1 Custom drive code is uploaded to 0200-$02ff.
A = 2 Custom drive code is uploaded to 0200-$02ff and the zeropage.
A = 3-7 Custom drive code is uploaded to 0200-$02ff plus 1-5 pages from $0300 upwards, and the zeropage.

Please note that if more than one page of custom code gets uploaded, then the zeropage gets always transferred last. This must be mirrored by the memory layout of the custom drive code in the C64's RAM (zeropage always last). Once the custom code transfer is complete, the plugin will execute it by issuing a JSR $0200. Therefore, the entry point of the custom drive code must be $0200, and it must exit with an RTS or JMP $0103 (if the stack pointer is altered).

Value of A	$0180-0$1ff	$0200-$02ff	$0300-$03ff	$0400-$04ff	$0500-$05ff	$0600-$06ff	$0700$07ff	ZP
0	---	---	---	---	---	---	---	---
1	*	B1	---	---	---	---	---	---
2	*	B1	---	---	---	---	---	B2
3	*	B1	B2	---	---	---	---	B3
4	*	B1	B2	B3	---	---	---	B4
5	*	B1	B2	B3	B4	---	---	B5
6	*	B1	B2	B3	B4	B5	---	B6
7	*	B1	B2	B3	B4	B5	B6	B7

*The order in which drive memory pages are occupied by custom code is determined by the value of A (--- = no upload; * = available for use; B1 – B7 = block order of uploaded drive code)*

3. Restoring Sparkle's drive code or resetting the drive

Sparkle_SendDrvCode (`JSR $0380`)

A = number of Sparkle drive code pages (#$00-#$06), same as in Sparkle_RcvDrvCode
X = C64 Sparkle store address high byte, same as in Sparkle_RcvDrvCode

After the custom code exits, we may want to restore Sparkle's drive code to resume normal loading operations. We will use the same function as we used to upload the custom drive code, but the values of A and X must be the same as what we provided for the Sparkle_RcvDrvCode call. If Sparkle is no longer needed after the custom code, then we may choose not to store and

restore its drive code. In this case the drive can be reset by calling this function with A = $FF, or directly from the custom drive code in which case this call can be omitted.

Here is an example code snippet using the custom drive code plugin:

```
lda #PlgIndex
jsr Sparkle_LoadA          //Load plugin with its directory index

lda #$06                   //6 pages: $0300-$07ff + ZP
ldx #>SparkleBuffer        //transfer and store Sparkle's drive
jsr Sparkle_RcvDrvCode     //code on the C64

lda #$07                   //7 blocks: $0200-$07ff + ZP
ldx #>DriveCode            //upload custom drive code
jsr Sparkle_SendDrvCode    //and execute it

//Run part here

lda #$06                   //parameters as in Sparkle_RcvDrvCode
ldx #>SparkleBuffer        //restore Sparkle's drive code and
jsr Sparkle_SendDrvCode    //continue standard loading operations

lda #BundleIndex           //Load next part
jsr Sparkle_LoadA          //must use index-based loader call
```

Another example in which we do not want to restore Sparkle after the custom code part:

```
lda #PlgIndex
jsr Sparkle_LoadA          //Load plugin with its directory index

lda #$00                   //0 pages, we are not storing
ldx #$00                   //Sparkle's drive code
jsr Sparkle_RcvDrvCode

lda #$07                   //7 pages: $0200-$07ff + ZP
ldx #>DriveCode            //upload custom drive code
jsr Sparkle_SendDrvCode    //and execute it

//Run part here

lda #$ff                   //reset drive
ldx #$00                   //omit this call if custom drive code
jsr Sparkle_SendDrvCode    //resets the drive directly
```

SPARKLE'S COMPRESSION AND DISK UTILIZATION

Sparkle uses its own limited 16-bit offset LZ-based compression algorithm optimized for speed. It combines the benefits of bitstream-based and byte aligned algorithms. Each block on the disk gets compressed on its own to allow random access and buffer-based decompression on the C64. The algorithm therefore primarily uses the data already compressed in the block as dictionary. Data in the same bundle stored on previous tracks is also used automatically to further improve compression of blocks of the same bundle stored on subsequent tracks. Additionally, Sparkle can use any static data in the RAM (loaded during previous loader calls or real-time generated) to further improve compression of the current bundle. The `Mem` entry type can be used to define these static memory segments. You must exert extreme caution in selecting static memory segments for a `Mem` entry. A single byte change in a `Mem` segment before the loader call may result in altered references and thus, errors during decompression.

When compressed bundles are added to the disk, most of the time the last block of a bundle would remain partially unoccupied. Sparkle, however, doesn't leave any disk space unused, and it will start adding the next bundle by filling first the unused space of the previous bundle's last block, before starting a new block. These transitional blocks contain the end of one bundle and the beginning of the next bundle simultaneously, and they are the only ones that are loaded in a fixed order (i.e., they are always the first/last blocks loaded). All other blocks of a bundle are loaded in a first come first serve order. Between loader calls, the transitional blocks occupy the buffer, and the next sequential loader call will first decompress the beginning of the next bundle from there, before loading the next block. This is why the loader buffer must be left untouched between sequential loader calls.

After each loader call Sparkle's drive code automatically prefetches the next block in sequence. So, when the next sequential loader call is issued, it will already have data ready to be transferred. If an index-based loader call is placed instead, then the loader will drop the prefetched (sequential) block and it will fetch the first (transitional) block of the requested bundle. This is why sequential loader calls are always faster than index-based loader calls. On the other hand, if we need to use the loader buffer for other purposes between loader calls, we can always issue an index-based loader call with the next bundle's index to reload the buffer.

COMMON ISSUES

Here is a check list to identify common pitfalls should you run into problems while using Sparkle.

1. Using `$dd00` to change VIC banks. Sparkle uses `$dd00` to communicate with the drive. Changing `$dd00` from your code may result in early reset of the drive or other unexpected loader activity and failing to load. VIC bank selection must be done by writing `#$3c-#$3f` to `$dd02` (format: `#$3c+VIC_bank`, where `VIC_bank = $dd00 value ^ #$03`).
2. Forgetting to restore `$dd02` to `#$3c+VIC_bank` before the next loader call. Some demo parts may write other values to `$dd02`. Having any other value left in `$dd02` when the next loader call is issued may result in early reset of the drive or other unexpected loader activity and failing to load.

3. Restoring the stack pointer to `#$ff`. This will result in overwriting the loader's resident code (`$0160-$02ff`) and thus, in a crash. Restore the stack pointer to `#$5f` instead.
4. Overwriting the loader buffer. The loader buffer (`$0300-$03ff`) contains the beginning of the next bundle between loader calls. Thus, it must be left untouched between sequential loader calls to allow Sparkle to depack the first partial block of the next bundle from there before loading commences. If you need to overwrite the buffer, then make sure the next loader call is bundle index based as it will reload the buffer.
5. Loading to pages 1-3. This will overwrite the loader's resident code or the buffer and result in a crash.
6. Failing to save, set and restore the value of the input/output register `$01` in the IRQ when loading under the I/O area. Sparkle sets `$01` to `#$35` at the beginning of each loader call but may change it to `#$34` if a bundle is being loaded to the RAM under the I/O area. Your IRQ must be able to handle this by saving the value of `$01` then setting it to `#$35` to enable accessing the I/O registers and finally, restoring its original value at the end of the IRQ.
7. The demo uses stacked IRQs (i.e., an IRQ interrupting another one) and it crashes during loading. You must make sure that you save the CPU registers (A, X, Y) and the status of the input/output register $01 to different addresses from the second, stacked IRQ, to avoid overwriting the first IRQ's values. The first IRQ saves the status of these registers when it interrupts the loading process and must return the same values for loading to continue properly. If the second, stacked IRQ overwrites these values by the ones it receives when it interrupts the first IRQ then these values will be passed to the loader instead of the original ones, resulting in a crash. Typically, the problem is using the same macros to start and finish both IRQs, thereby saving the registers to and restoring them from the same ZP address. There are several different ways to avoid this. E.g., save and restore the registers to and from the stack. Or, if you save them to the ZP, you must use different ZP addresses for the stacked IRQ.
8. Failing to make sure Sparkle's zeropage addresses are not used by any other process during loading. You can always save and restore these zeropage addresses in your IRQ or change Sparkle's ZP usage in your script.
9. The demo works in one emulator, but it is out of sync/stalls in another one or when loaded from a real floppy disk. Loading speeds may differ slightly depending on the media the demo is loaded from. There can be also differences between drives and even when using the same drive with the same disk. On real disks tracks are not aligned (i.e., the first sectors of adjacent tracks are not positioned at the same angle of rotation around the disk) and the amount of skew depends on the formatting tool used. When formatting of a track is finished, the R/W head jumps to the next track and it takes some time for the tool before it starts writing out the next track all the while the disk keeps spinning. There can also be subtle variances in the size of the gaps between sectors on the formatted disks. All in all, these features may result in up to one full rotation (10 frames) difference in fetching the last sector of a file/bundle. Add the possibility of a failed checksum (which may happen on real disks, especially under party compo conditions) which will also require a full rotation before the loader can refetch the missed sector. Therefore, it is recommended to calculate with at least a 10-frame sync buffer, but preferably 20 (or more) after loading when a frame counter is used to sync the effects with the music. It is also important to use a sync code that allows the demo to continue if a sync frame is missed (using

BCC/BCS for comparisons instead of BEQ/BNE). Otherwise, a missed sync frame may result in a 16-bit wraparound delay.

10. All the file parameters in a bundle seem to be correct in a manually edited script and even I/O status of the file chunks is correct but the bundle still fails to load correctly. Make sure the file parameters are separated by TAB characters or place the file name in double quotes to allow the use of space as parameter separator.
11. Files are corrupted after using the `Mem` entry to define memory segments that can be used to improve compression of a bundle. Make sure the `Mem` entry describes a static memory segment that remains unchanged during loading and decompression of the bundle.

DISCLAIMER

Sparkle is free software and is provided “as is”. It is a hobby project of a hobby coder so use it at your own risk and expect bugs. I do not accept responsibility for any omissions, data loss or other damage. Please credit me in your production should you decide to use Sparkle or any pieces of it. Feel free to contact me with any questions via PM on CSDb or by emailing me at spartaofomgATgmailDOTcom.

Please find the most up-to-date version of Sparkle here: https://github.com/spartaomg/SparkleCPP

Sparta, 2019-2025

VERSION HISTORY

V3.2

- Reliability and stability improvements without compromising speed. Added header block sanity checks and re-added trailing zero first nibble check. Improved track correction code: if the R/W head lands on the wrong track, then Sparkle will correct it (in part adopted from Krill). Minor GCR loop adjustment for better high rotation speed tolerance in Zone 2. Loop tolerates at least 272-314 rpm in all 4 speed zones. Simplified GCR loop patch code (patch tables on page 3). Simplified track seek code by eliminating early track change during sequential loading (stepping before ATN check).

- Reworked plugin handling and syntax, allowing multiple plugins and hi-score files per disk side. Custom code and Saver plugins are no longer mutually exclusive. Plugin (`Plugin: custom` for custom code and `Plugin: saver` for hi-score file saver) and hi-score file entries are now to be added after the Disk info section of the script as separate bundles, before or after all other file bundles. Each hi-score file entry must be preceded by its own saver plugin (in a separate bundle). Plugins and hi-score files are located after the last regular file bundle on the disk, and they can only be loaded using index-based loader calls. The new `PlgIndex` entry type is introduced to assign specific directory entry indices to plugin and hi-score file entries. Bundle/directory indices #$7e-#$7f are no longer reserved for this purpose. The use of `PlgIndex` or `DirIndex` entries is highly recommended. Hi-score files are no longer limited in size (as long as they don't need more than 255 blocks on the disk) and can span over multiple tracks.

- Bug fix: stop with error message if a file memory segment end address is less than the start address with the "-" file parameter syntax.

- Bug fix: if custom drive code required 1 block in the drive's RAM, then it was erroneously loaded to ZP instead of $0200.

- Bug fix: if the `DirIndex` entry preceded the first `File` or `Mem` entry in a bundle then it was added to the previous bundle.

V3.1

- New optional command line parameter: `-p [a/e]` to pause on exit. Use this parameter if you want Sparkle to pause on exit and wait for `Enter` to be pressed before finishing. If you only want to pause on error, then use it with the value `e`. To always pause on exit, use it with the value `a`. The `-p` command line parameter must always be preceded by the script file name.

- Alternative `File`, `HSFile`, and `Mem` entry type syntax. Space is now allowed as file parameter separator if the file name is placed between double quotation marks (as in "path/part.prg"). If double quotes are not used, then only the tabulator character is accepted as file parameter separator, as in previous versions.

- Alternative file parameter syntax for PRG files and other file types with a 2-byte PRG header. To use this syntax, the first file parameter must be "-" (hyphen), the second one is the memory address of the first byte, and the third one is the address of the last byte to be loaded.

- Put the file name in {curly brackets} instead of double quotes if you want to have them added to the disk uncompressed. Curly brackets also enable the use of space as file parameter separator.

- Multi-side D64s are now automatically marked with `_SideA/B` etc. if the `Path` entry with the D64 file names of the sides is omitted in the script.

- Custom drive code plugin. Adding the `Plugin: custom` entry to the Disk Info section of the script will instruct Sparkle to add the plugin to the disk. It uses the last two blocks of the disk and can be evoked with bundle index #$7e (similar to the hi-score saver plugin). To free the drive's RAM, part or all of Sparkle's drive code must be transferred and stored on the C64 (max. 6 blocks needed). The plugin uses $0100-$017f after the code transfer, so the entire zeropage + $0180-$07ff are available on the drive if needed. Once the custom drive code is no longer needed, Sparkle's drive code can be restored from the C64, or the drive can be reset.

- Updated bootstrap code to allow "Mount & Run" on the Ultimate family and proper loading of the drive code even if the drive is reset or power cycled between BASIC commands LOAD and RUN.

- Sparkle's installer no longer sets the NMI vector to an RTI instruction.

- Bugfix: loading to the zeropage was handled by the same code as loading to the shadow RAM under the I/O registers. Because of this, in previous versions, Sparkle wrote $34 to data input/output register $01 in both cases, thereby turning I/O off also while loading to the zeropage, which could lead to a crash if the IRQ was not prepared to handle this. This issue is now fixed and loading to the zeropage will not affect the status of $01.

- Thanks to hedning/G*P for testing, Raistlin/G*P for help and testing, and Wacek/Arise for suggestions and testing!

V3

- Full rewrite in C++, allowing cross-platform use. The Sparkle PC tool has been tested on Windows 11, Intel macOS Monterey 12.7, Ubuntu 22.04, as well as WSL Ubuntu, Debian, and Kali Linux. This version is a command-line tool, lacking the previous versions' GUI.

- New memory layout of the C64 resident code. The loader occupies $0160-$02ff and the loader buffer $0300-$03ff. The stack is reduced to $0100-$015f. The loader code now starts with the fallback IRQ which is followed by the Sparkle_SendCmd function and then by the code needed for Sparkle_LoadNext calls. If the fallback IRQ is not used, then stack can be extended to $0100-$017a. If neither the fallback IRQ nor LoadA calls are needed, then the stack can be extended to $0100-$019f. Simply update the stack pointer using the LDX #$7a/#$9f + TXS instructions.

- Improved file compression with limited 16-bit match sequence offsets. Data blocks are loaded and decompressed in a first come first serve order so normally they cannot rely on data outside their own boundaries (i.e., the content of other blocks in the same bundle). This limits compression efficacy

because there is no guarantee those blocks will get loaded ahead of the current one. There are, however, exceptions. Blocks containing the end of one bundle and the beginning of the next one are always loaded in a fixed order. Blocks from previous tracks can also be relied upon because the R/W head only moves to the next track once all blocks from the current one got loaded. Sparkle takes these variables into account during compression. The user can also specify previously loaded or runtime generated static memory segments using the newly introduced `Mem` entry type that Sparkle will also use to search for match sequences outside the block's boundaries to further improve compression.

- File parameter math expression evaluation. Sparkle uses the TinyExpr++ library by Blake Madden (which is based on the TinyExpr library by Lewis Van Winkle) to evaluate math formulas. Formulas must start with a = sign followed by the formula. As previously for file parameters, the default number format is hexadecimal and any number without a prefix is regarded as a hex number. For decimal numbers, the "." prefix must be used. For example, the following entry

```
File: koala.kla  d800  =2+.8000+.1000   =.200+(2*.400)
```

will load 1000 ($3e8) bytes to the color RAM from decimal offset 9002 ($232a) in koala.kla. The use of constants is not supported.

- Directory art import: Sparkle now supports import from D64, PRG, TXT, BIN, C, PET, KickAss ASM, JSON, PNG and BMP image files. Sparkle uses the LodePNG library by Lode Vandevenne to decode PNG files. Image files must use two colors and must have a width of exactly 16 characters (128 pixels or a multiple of 128 if the image is enlarged) and a height of a multiple of 8 pixels. Borders are not allowed. Sparkle will attempt to find a space character in the image to determine background and foreground colors. If no space is found, then it will use the darker color as background. Image files must display the uppercase charset, and their appearance cannot rely on command characters.

- Introducing the `DirIndex` bundle entry type. By default, all bundles (or up to 128) are added to Sparkle's internal directory (as in all previous versions of Sparkle). By adding the `DirIndex` entry type to at least one bundle the user can create an alternative internal directory structure where only bundles with a `DirIndex` are added to the internal directory. The `DirIndex` entry can have a random value between $01-$7f (or $01-$7d if a hi-score file is added to the disk). `DirIndex` value $00 is preserved for bundle 0 and is used internally. For this reason, bundle 0 doesn't require a `DirIndex` entry. If at least one `DirIndex` is added to the script, then all other bundles (the ones without a `DirIndex`) will only be accessible using a sequential (Sparkle_LoadNext) loader call. `DirIndex` values are constant and will remain the same even if the order of bundles changes.

- Introducing the `ThisSide` and `NextSide` disk info entry types. They are only needed if disks from a multi-disk project are created separately, from their own scripts. They can take a hex value between $00-$7e. If defined, the values of these entries will be used to identify the sequence of disks. If the disks are created using a single script, then these entries are not needed and Sparkle will assign a disk index to each disk automatically, starting with 0 for the first disk, and the index is autoincremented for each subsequent disk.

- The `Sparkle_InstallIRQ` and `Sparkle_RestoreIRQ` loader functions have been removed due to memory constraints.

- Bug fixes.

- Thanks to Ksubi/G*P, Raistlin/G*P, Visage/Lethargy, Loloke/Lethargy, Schedar/Lethargy, Soci/Sigular for testing and help, and Magnar/Censor Design for suggestions!

V2.2

- Decimal number format support for the file parameters. If you manually edit your script, use the "." prefix for decimal numbers as shown in the following example:

```
File: myfile.bin       .16384     0000  .8192
```

In this example the first and last file parameters (load address and length) are decimal numbers while the second parameter with no prefix (offset) is hexadecimal. Decimal number format can be also used for the hi-score file. In the Editor window, press the "." key in the file parameter fields to switch to decimal mode. To return to hexadecimal mode either press the "H" key or simply leave the edited field and Sparkle will automatically convert its content to hexadecimal.

- Sparkle now allows the `Header`, `ID`, and `Name` entries to be left blank and will not use a default value. If the `Name` entry is not defined and a `DirArt` file is attached to the script, then Sparkle will use the first entry in the `DirArt` file as the main directory entry on the disk. If the `DirArt` is imported from a D64 file, then Sparkle will also import the `Header` and the disk `ID` from the `DirArt` file if these are not specified in the script.

- Slight improvement in compression efficacy.

- Stability improvements and bug fixes.

- Thanks to Visage/Lethargy and Grass/Lethargy for testing, and Wacek/Arise for suggestions!

V2.1

- Full rewrite of the GCR loop resulting in a much wider disk rotation speed tolerance of at least 269-314 rpm across all 4 speed zones. Checksum verification happens on-the-fly for disk zones 0-2 (tracks 18+) while it is done partially outside the GCR loop for zone 3 (tracks 1-17). Since the native interleave of this zone is 4 but fetching and transferring a block requires only a little bit more than 3 sectors passing under the RW head, there is plenty of time left to finish checksum verification outside the loop without a performance penalty.

- Reintroduced the second block buffer feature which was first invented for Sparkle 1.x but was dropped in Sparkle 2 due to drive memory constraints. Rewriting the GCR loop allowed freeing enough memory in the drive's RAM for this feature, so Sparkle 2.1's speed is now on par with Sparkle 1.5.

- Implemented an ATNA-based transfer loop.

- Added full block ID check to improve reliability and to avoid false data blocks on Star Commander warp disks.

- Commenting the script. In the Editor, there are two options to add comments. Each file entry has a File Comment subentry for file specific comments. The other option is adding Comment entries to the script

that can be placed anywhere between Disk, Bundle the Script entries. Comment entries cannot be inserted as subentries into Disk, Bundle or Script entries. If the script is manually edited in a text editor, comments can be added anywhere as long as they are separated by one or more tabs from the rest of text in the line or are placed in separate text lines. *Sparkle handles everything as a comment it doesn't recognize as a standard script entry.* The user may choose their preferred comment identifier or use nothing as long as the comment cannot be misinterpreted as a script entry. When the script is saved from the Editor, Sparkle will use the `#comment` format. Comment identifiers are only visual aids and are not used to determine whether an entry is a comment or not.

- Other stability improvements and bug fixes.

- Thanks to hedning/G*P, KAL_123/Miami Fun Project, Raistlin/G*P, and Visage/Lethargy for testing, Rico/Pretzel Logic for bug report, Wacek/Arise for suggestions and bug report, and Ksubi/G*P for help!

V2

- Sparkle now supports random file access, i.e., bundle index-based loading. The bundle index must be loaded in A. Two sectors are used as internal directory on track 18 and each directory entry needs 4 bytes. Thus, a maximum of 128 bundles can be loaded by index (`$00-$7f`). Note that there can be more than 128 bundles on the disk, but only the first 128 can be accessed this way. Subsequent bundles can only be loaded sequentially, with a Sparkle_LoadNext call. Calling the loader with indices `$80-$fe` in A is interpreted by Sparkle as a disk flip call (subtract `$80` for the corresponding disk index). `A=$ff` is used with the Sparkle_SendCmd function to reset the drive. Bundle indices `$7e` and `$7f` will load the saver code and the hi-score file, respectively, if a hi-score file is included on the disk (see below).

- "Hi-score file" saver with limited file saving capability. Sparkle can now overwrite an existing file on the last track of the disk (track 35 on a standard disk and track 40 on an extended disk). The hi-score file can be specified in the disk info section of the script. File size must be `$100-$f00` bytes and it must be multiples of `$100` (i.e., the low byte of the file size must be zero). If a hi-score file is added to the script, Sparkle will include the saver code on the last track of the disk with an index of `$7e` while the hi-score file's index will be `$7f`. The saver code and the hi-score file can be only accessed using index-based loading. To overwrite the hi-score file, first load the saver code then call it with the high byte of the file size in A. To exit the saver function without saving anything, call it with `$00` in A. The saved file can be smaller, but it cannot be larger than the original hi-score file. Sparkle can also save from the RAM under the I/O area (`$d000-$dfff`).

- Product ID: a 3-byte long unique identifier (max. 6 hex digits) that is used to identify disk sides belonging to the same product/build. The Product ID is shared between all disks built from the same script. This will ensure that disks of Product 2 will not be accepted by Product 1. If a Product ID is not specified by the user, then Sparkle will generate a pseudorandom number every time the script is run.

- NTSC support.

- 40-track disk support. Tracks 36-40 are extensions of tracks 31-35 (17 sectors each, speed zone 0).

- Loop feature has been removed.

- Huge thanks to d'Avid/Lethargy, hedning/G*P, Raistlin/G*P, and Visage/Lethargy for testing and Ksubi/G*P for help!

V1.5

- Updated compression algorithm resulting in about 20% faster decompression with only about 0.3-0.5% decrease in compression efficiency. This typically means no more than 3-4 extra blocks per disk side while loading is faster, especially under heavy processor use as Sparkle spends significantly less time depacking.

- Sparkle can now handle not only TXT but also D64, BIN, and PRG DirArt file formats.

- IRQ Installer moved to `$01d5`. If you want to install the Fallback IRQ without changing the subroutine call in it then call `jsr $01db`.

- Bug fixes. Thanks to Visage/Lethargy and Schedar/Lethargy for reporting bugs and testing.

V1.4

- New feature: Sparkle shows a warning if there are multiple active drives on the serial bus. The demo will continue once all devices but one are turned off. Thanks to Dr. Science/ATL for this feature request.

- Removed optional packer selection. Sparkle now uses an updated version of its former "better" packer with an optimized decompression algorithm.

- Updated GCR loop for increased stability. Sparkle now has a disk rotation speed tolerance of at least 284-311 rpm across all four speed zones.

- Loader "parts" are renamed to file bundles to avoid confusion.

- The disk monitor now highlights the `[00 F8]` file bundle separator sequence.

- Bug fix: the editor did not calculate bundle and disk sizes correctly.

- Other minor bug fixes and improvements.

V1.3

- New feature: script embedding. If your script is very long, you can save part of it in a separate file and then add this to your script using the "`Script:`" entry type followed by <TAB> and the script file's path. When Sparkle reaches a script entry during disk building, it will first process its content before continuing with the next entry. You can even add whole disks to your script this way. Scripts cannot be inserted in an existing file bundle. I.e., files in the embedded script will always start a new file bundle and won't be added to the current bundle. If relative paths are used, Sparkle will use the path of the embedded script to calculate the path of the files in it.

- New feature: demo looping. Use the "`Loop:`" entry type followed by <TAB> and a decimal value between 0-255 in the *first* disk's info section to determine your demo's behavior once it reaches its end. The default value is 0 which will terminate the demo. A value between 1-255 will be interpreted as a disk number where 1 represents the first demo disk. Once the last bundle on the last disk is loaded Sparkle will wait for this disk to be inserted to continue. If you use the last disk's number, then Sparkle will reload the last disk in an endless loop. This entry type can only be used once in a script. If not specified, then the default value of 0 will be used.

- New feature: aligning a bundle with a new sector on the disk. By default, Sparkle compresses files back-to-back, not leaving any unused space on the disk. If the length of a bundle changes during demo development, it will affect the compression and distribution of every subsequent bundle on the disk. This may adversely influence the timing of the demo. From the editor, double-click the "Start this bundle in a new sector on the disk" line under the bundle node to change its value to YES from NO where this type of timing is crucial to force Sparkle to always start the bundle in a new sector on the disk. If you prefer to manually edit your script, use the "`Align`" entry type in a new line preceding a bundle to achieve the same result.

- New feature: custom sector interleave. Sparkle now allows the user to specify the interleave for all four speed zones on the disk. The default is 4 for tracks 1-17 (IL0), and 3 for the rest of the disk (tracks 18-35, IL1-IL3). Use `ILn:<TAB>N` in the disk info section in your script where n=0-3 specifies the zone and N>0 is a decimal value for the desired interleave to be used during disk building.

- New feature: Sparkle now generates an exit code when running from command line. The exit code is non-0 if there is an error during disk building.

- Improved and updated editor to accommodate the new features. The editor now accepts alphanumeric (a-z, 0-9) characters in addition to <Enter> to start editing an entry. Just press the first character of the new value to overwrite the previous one, or <Enter> if the first character is not alphanumeric. Once you are done editing, press <Enter> or <Down> to step to the next entry, or <Up> to step back to the previous one.

- Updated, more flexible script handling:

    - Sparkle now recognizes both LF and CRLF line endings.

    - The script entry "`New Disk`" is no longer needed to start a new disk during manual script editing. Just add the next disk's info after the last file or script entry. Make sure there is at least one file entry after every disk info section. Otherwise, the next disk info section will overwrite the previous one.

    - Sparkle will skip any unrecognized lines in the script. This can be used to manually comment your script. Manual comments will be ignored in the editor window and will be lost when the script is resaved from the editor.

    - File offset values can be as large as `$ffffffff`. The maximum value of file address and length remains `$ffff`.

- Bug fixes. Thanks to Raistlin/G*P and Visage/Lethargy for testing and feature requests.

V1.2

- Optional better packer. Sparkle now offers two versions of its packer. The original, faster one, and a new, better one. The new packer results in better compression at the expense of slower packing and depacking compared to the original faster but less effective option.

- Minor improvements in the C64 code saving about 10000 cycles on the depacking of a disk side.

- GUI update.

- Bug fixes.

V1.1

- Option to select ZP usage in the script.

- Improved default file parameter handling.

- Minor changes in the editor.

- Bug fixes related to loading under I/O.

V1.0

- Initial release.