SPARKLE 2.1

Table of Contents

INTRODUCTION

Sparkle is an IRQ loader and linking solution for the Commodore 64 inspired by the loaders of Lft, Krill, and Bitbreaker. It utilizes on-the-fly GCR processing, fast data transfer, and blockwise data compression. Disks are built using loader scripts, files are bundled together and are loaded in batches. Sparkle handles multi-disk projects and offers limited file saving capability. A Windows tool is provided to edit script files and build Sparkle disks. For version history and description of new features please see the Appendix.

SUMMARY OF FEATURES

-   Tested on 1541, 1541-II, 1571, and Oceanic drives and the 1541 Ultimate family.
-   Resident size: $2a0 bytes including loader, depacker, fallback IRQ ($0160-$02ff), and buffer ($0300-$03ff). Stack is reduced to $0100-015f. The buffer contains preloaded data between loader calls, so it needs to be left untouched for sequential loading.
-   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 269-314 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 loop in zone 3.
-   C64 reset detection.
-   2Bit+ATN transfer protocol, 72 bycles/block 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.
-   Built-in blockwise packer/depacker. The packer compresses file bundles back-to-back. Thus, 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. The user can freely alter $dd02 between loader calls, as long as its value is restored before loading. $dd00 needs to be left untouched.
-   Hi-score file saver. Sparkle is able to save a single file of max $0f00 bytes overwriting a pre-defined hi-score file. Saving is also freely interruptible.
-   Loading and saving under I/O.
-   40-track disk support adding 85 sectors to the standard 35-track disk.
-   PAL and NTSC support.

SPARKLE WINDOWS TOOL

The Sparkle Windows tool (written in VB.NET, target .NET Framework 4.8) features a simple disk monitor and a built-in script editor. D64 and script files can be opened from within the tool or drag-and-dropped to process them. Script files use the .sls (Sparkle Loader Script) extension. Run Sparkle as administrator to associate the .sls file extension with the tool. Once the necessary registry entries are created you can also build your Sparkle disks by double-clicking script files. (This can also be achieved by selecting Sparkle from the Open with... list after double-clicking a script file.) Sparkle can also be used as a command line tool (e.g. sparkle mydemo.sls). A simple demo project is provided as an example.

MAIN WINDOW

Sparkle features a simple disk monitor which is the main window of the program showing the hex and PETSCII views of the selected sector and two toolbars. Here you can start a new disk ⬛, open 📂, and save 💾 🖫 D64 files, and build demo disks from scripts. The ⬛ button allows adding a standard (non-Sparkle) file such as a noter file to a previously built Sparkle disk. On the second toolbar, the leftmost two buttons will show the BAM ⬛ and the first sector of the directory ⬛. The next four buttons will navigate to the first ⬛, previous ▲, next ▼, and last ⬛ track on the disk. The following four buttons will load the first ⏮, previous ◀, next ▶, or last ⏭ sector of the selected track. The next group of six buttons are used to navigate through file bundles, jumping to the first sector of the first bundle ⏮⏮, the first sector of the previous bundle ◀◀, the previous sector in the bundle ➖, the next sector in the bundle ➕, the first sector of the next bundle ▶▶, and the first sector of the last bundle ⏭⏭. Finally, on the right side of the toolbar you can see the current or manually select the desired track T: 18 and sector S: 1 . Hovering over a button will show a tooltip with the description of the button's function and its shortcut key. In the hex view panel, you can perform changes in the selected sector.
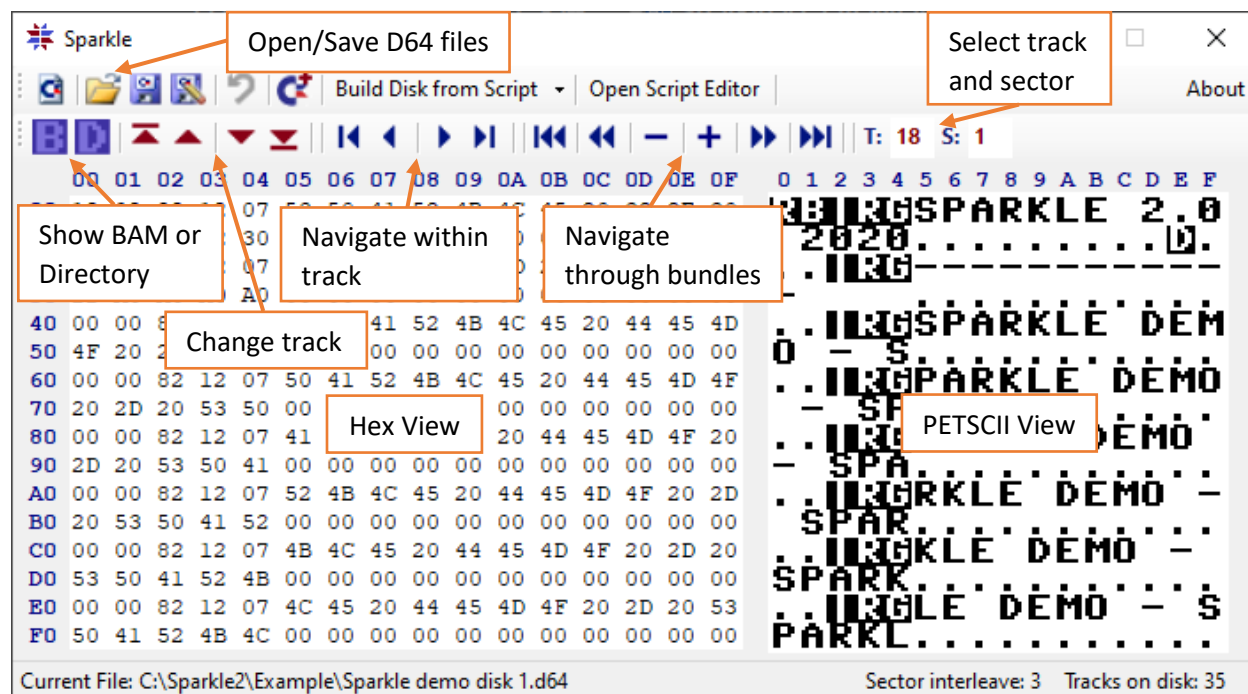
SCRIPT EDITOR WINDOW

Scripts can be created, loaded and saved in the script editor window where demo information is organized in a treeview structure. Script entries can be edited by double clicking or selecting them and pressing <Enter>. This will initiate editing by selecting the editable text portion of the entry or opening a dialog window where a file path is needed. Editing entries that accept free text can also be initiated by pressing an alphanumeric (a-z + 0-9) key which will immediately overwrite the entry's last value. Pressing <Escape> will cancel editing and restore the entry's previous value. To finish editing, press <Tab>, <Enter>, <Down> to move selection down or <Up> to move selection up. Entries that have a default value can be reset by deleting their current value and then leaving the entry. DirArt files, disks, file bundles, demo files, and embedded scripts can be deleted using the <Delete> key. The editor marks default parameters with lighter color then modified ones.

The top line in the editor (This Script) shows the current script's path and file name. If you start a new script, it will be updated once you save it. You can also double-click this entry (or press <Enter>) to load a script. The Add… entry and its four subentries (New Disk, New Bundle, New Script, New Comment) are always located at the bottom of the script and can be used to add these entry types to the script.
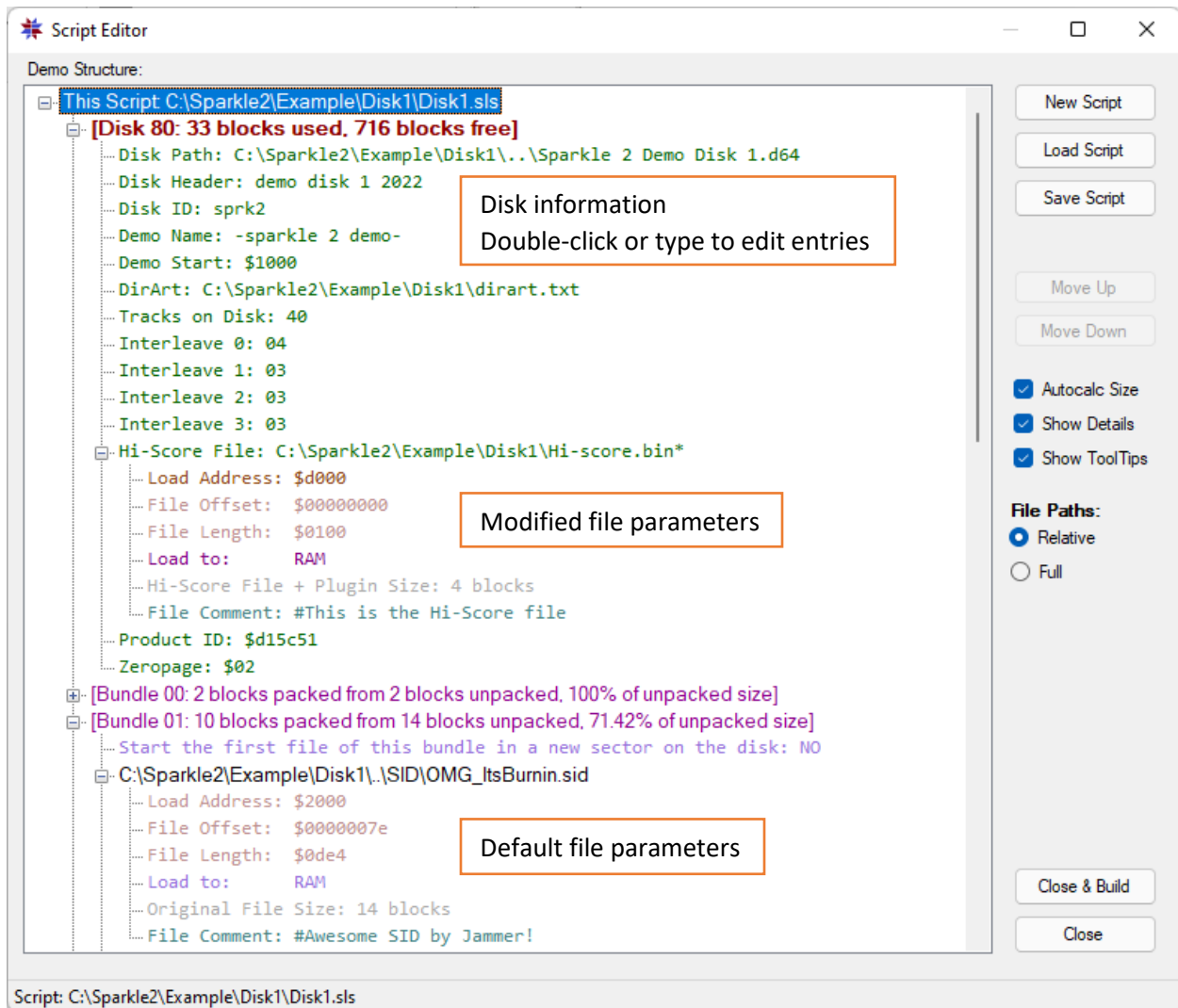
DISK INFO

To build a disk, open the editor and first add a blank disk structure (Add… -> New Disk). In the disk structure you can specify all necessary settings related to this disk. Zeropage usage and the Product ID are global settings and can be only specified once, in the first disk. The following options are available:

Disk Path: Here you can specify the final D64 file's name and path. Sparkle will use this information when the demo disks are created.

Disk Header: Max. 16 characters that will be used as the disk's header information in the topmost line of the disk's directory structure.

Disk ID: Max. 5 characters that will be located on the right side of the disk header in the directory structure.

Demo Name: Max. 16 characters. This will be your first directory entry on the disk. Sparkle will load the installer when this entry is loaded.

Demo Start: The program entry of the demo when loaded from BASIC. 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 entries, including the one specified at the Demo Name entry type). Each DirArt directory entry is executable and will start the demo but only the very first one will have a non-0 block size. Sparkle accepts 4 different file formats.

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.

D64 files: Sparkle will simply import the directory structure of the D64 file. It will import every entry type (DEL, SEQ, etc.) as PRG. The imported DirArt entries will have 0 block lengths.

PRG files: prepare your DirArt design on your C64's screen starting at the top left corner and save $0400-$07e8 (or as many screen rows as entries your DirArt has) as a PRG. Sparkle will ignore the first 2 address bytes of the PRG file and use the first 16 characters in every 40 as a DirArt entry.

BIN files: essentially the same as the PRG type less the address bytes.

Tracks on Disk: 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.

Interleave 0-3: These entries specify the distance in sectors between two consecutive data blocks on the disk. The default values are 4 for tracks 1-17 (Interleave 0), and 3 for tracks 18-24 (Interleave 1), 25-30 (Interleave 2), and 31+ (Interleave 3). 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 (interleave MOD number of sectors per track 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 demo part. Once the loading sequence enters a new speed zone on the disk, a new interleave can be used.

Hi-score File: You can add a default hi-score file to the script here that will instruct Sparkle to include the file saver code on the disk. The saver code and the hi-score file will be written to the last sectors of the last track on the disk (track 35 or 40 depending on disk type) with bundle indices $7e and $7f, respectively. The hi-score file's size must be multiples of $100 and it cannot be larger than $f00 bytes. If this script entry is left blank then the saver code will not be included on the disk and bundle indices $7e and $7f will be used as standard Sparkle bundle indices. Sparkle supports saving from under the I/O area.

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 can only 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.

Zeropage: Sparkle uses 3 bytes in the zeropage. The default is $02-$04. 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 can only be specified once, in the first disk, and it will be used for the rest of the demo. 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

After completing the disk info section, you can start adding files to your disk. Instead of loading files one by one, Sparkle bundles them together and loads 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 (as long as they do not overlap in the memory) and load them in one call. 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 demo 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 at the same memory address.

To add files to your demo disk, first you need to create a new bundle (Add… -> New Bundle). Once this is done, you will see two new entries under the bundle entry. The first one ("Start the first file…") is a bundle-specific setting 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. I.e., a new bundle will first occupy the unused space in the last sector of the last bundle before starting a new block in the next 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. This behavior can be changed by double-clicking the "Start the first file of this bundle in a new sector on the disk" entry which will toggle its value between YES and NO. Changing the value to YES will insert the `Align` key word in the script file in the line preceding the first file of the bundle which will force Sparkle to leave the last sector of the previous bundle partially unused and start the new bundle in a new sector on the disk (similar to how other loaders work). This can be helpful in loading time sensitive demo parts. However, most of the time, this option would just inflate disk usage.

When adding files to a bundle, you can specify what and where you want to load by entering the load address (where the data will go), offset (first byte of the file to be loaded), and length of the desired data segment (number of bytes to be loaded) within the selected file. During disk building, Sparkle will only compress the selected segment of the file. By default, Sparkle assumes that the file is a PRG independently of the file extension and uses the first two bytes of the file as load address, 2 as offset, and (file length-2) as length. The only exception is SID files where file parameters are extracted from the file automatically. If you change the file's load address then Sparkle will presume that the file is NOT a PRG and it will change the offset to 0 and will use the file's full length as length, accordingly. Thus, you will need to make sure that the offset and file length are correct after changing the load address.

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 by double-clicking the "Load to" entry under the file. If the file is destined in the RAM under the I/O area Sparkle will 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 files within a bundle during compression to achieve the best possible compression ratio. Therefore, file order within a bundle can be random.

SCRIPT ENTRIES

Demo scripts can become very long, so you may want to create shorter scripts, and then include these in your main script (Add… -> New Script). You can add whole disks to your script this way. If a script is saved using relative file paths, then the script's path will be used to calculate file destinations when it is embedded in another script. When Sparkle reaches a script entry during disk building, it will first process its content before continuing with the next entry. Scripts cannot be inserted into 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. The content of embedded scripts cannot be edited from their parent script, only separately, the way they were created.

Disks, bundles, embedded scripts, and comment entries can be freely rearranged. If a disk info section does not precede the first bundle, then Sparkle will use default disk parameters for the first disk (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). If two disk info sections follow each other without any demo files between them then the second one will overwrite the values of the first one during disk building.

Once your script is finished, you can build your disk by clicking the "Close & Build" button. The script remains active until another one is loaded or created. Opening the editor window again will show the active script.

MANUAL SCRIPT EDITING

Use the following template to manually prepare or edit your scripts:

```
[Sparkle Loader Script]

Path:       path\file.d64
Header:     max. 16 characters as in the directory header
ID:         max. 5 characters as in the directory ID
Name:       max. 16 characters
Start:      xxxx  (program entry when the disk is started from Basic)
DirArt:     path\file
Tracks:     35 (default) or 40
IL0:        n (sector interleave of n>0 for tracks 1-17)
HSFile:     path\file*      xxxx  yyyyyyyy   zzzz
ProdID:     abcdef (6-digit hex number)
ZP:         xx (set once in first disk info section)


File:       path\file*      xxxx  yyyyyyyy   zzzz
File:       path\file       xxxx  yyyyyyyy   zzzz


Align
Script:     path\script
```

The script must start with the file type identifier in brackets and must contain at least one `File` entry. Everything else is optional and can be omitted. If `Start` is omitted Sparkle will use the load address of the first `File` entry as start address. Entry types (`Path, Header, ID, Name, Start, DirArt, Tracks, IL0-IL3, HSFile, ProdID, ZP, File, Script`) and their values must be separated by one or more tabs. The loader's zeropage usage (hex byte format, default is `02`) and the Product ID can only be set once, in the first disk's setup. File bundles are separated by blank lines. Files in a single bundle are in consecutive lines. File paths can be absolute or relative to the script's path. After at least one `File` or `Script` entry is added to the script, you can start a new disk simply by adding new disk info entry types (`Path, Header`, etc.) followed by the next disk's files. See example demo project's script for details.

FILE PARAMETERS

You can specify three parameters for each file entry, separated by tabs. The first one 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. Parameters are hex numbers 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. 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. 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. Sparkle will calculate the length as (original file's length – offset), but max. ($10000 - load address) if the load address and the offset are given but the length is not. The "Align" entry type is used in a new line preceding a bundle or script entry to align it with a new sector on the disk (see the Script Editor section for details).

LOADING UNDER I/O

By default, Sparkle writes #$35 to $01 and turns the BASIC and KERNAL ROMs off during loading but leaves I/O on. If any part of a file is to be loaded in the RAM under the I/O area ($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, 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.

COMMENTING

Sparkle allows commenting the script and in general, will interpret any unrecognized entries as comments. In the Editor window, the user has two options to add comments to the script. Each file entry has a File Comment subentry where a file specific comment can be inserted. The other option is adding separate Comment entries to the script (Add… -> New Comment). These can be placed between Disk, Bundle, and Script entries. Comment entries cannot be inserted as subentries into Disk, Bundle or Script entries. If the script is edited manually in a text editor, comments can be added essentially anywhere as long as they are in a separate text line or separated by one or more tabs from the entry in a text line:

```
#Bundle comment
File:      path\file   xxxx  yyyyyyyy   zzzz  #file comment
File:      path\file   xxxx                   #file comment
```

You can choose the comment identifier of your preference or use none as long as the comment cannot be misinterpreted as a standard entry. When the script is saved from the Editor window, Sparkle will use the #comment format. Note that the # sign is only a visual aid. It is not used to determine whether an entry is a comment or not. If a manually edited script is opened in the Editor window, it will only recognize and display comments in separate lines and after file entries. Any other comments will be ignored and will be deleted if the script gets resaved from the Editor window.

RUNTIME CONSIDERATIONS

The installer, C64 resident code, drive code, and custom directory take 12 blocks on track 18. Thus, you have the entire 664 blocks for your demo and the remaining 6 blocks of track 18 for DirArt. Loading and running *any* entry from the disk's directory will start the installer which will install the C64 resident code

and the drive code, set the I flag, write #$35 to $01, reduce the stack to the lower $60 bytes and set the NMI vector to an `rti` instruction. Then the installer will automatically load the first file bundle. During decompression, the value of $01 can be either #$34 or #$35 depending on the destination of the files in the bundle. Once the first bundle is loaded, the loader will restore $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. The loader writes #$35 to $01 at the beginning of every loader call and will return with this value in $01. Loader calls do not clobber the I flag.

LOADER FUNCTIONS

From your code, the following functions are available:

- Sparkle_LoadA (JSR $184)
  - *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. 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. To request another disk side, use indices $80-$fe in A ($ff is reserved for drive reset) where $80 is the index of the first disk in the script. 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_LoadNext (JSR $1f9)
  - *Sequential loader call, parameterless, loads next bundle in sequence*

  This parameterless call will load the next bundle of files as specified in the script. When the loader is called, it first depacks the first partial block (if such exists) of the next bundle from the buffer before receiving the next block from the disk. Thus, 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.

- Sparkle_SendCmd (`JSR $160`)
    `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 $187`)
    Loads a prefetched bundle, use only after fetching a bundle using Sparkle_SendCmd (A=bundle_index). See example above.

- Sparkle_InstallIRQ (`JSR $2d2`)
    Installs fallback IRQ (A=raster line, X/Y=subroutine/music player vector high/low bytes)

    Use this function to install a simple fallback IRQ with a music player call or any other function. The IRQ routine is located at `$02e6-$02ff` and the subroutine call in it initially points at an `rts` instruction. The I flag is set during the initialization of the loader and remains set after loading the first bundle is completed. Calling the IRQ installer does not change the I flag allowing the user to set or clear it at the desired moment. You need to initialize raster IRQs and update all the necessary registers before installing the fallback IRQ as the installer will only set `$d012` and the `$fffe/$ffff` IRQ vector.

    ```
    ldy #<Player                //IRQ subroutine vector Hi Byte
    ldx #>Player                //IRQ subroutine vector Lo Byte
    ```

```
        lda #Raster                //Value for $d012 in A
        jsr Sparkle_InstallIRQ     //Enable fallback IRQ
```

- Sparkle_RestoreIRQ (JSR $2d8)

    Re-enables fallback IRQ without changing the subroutine vector (A=raster line).
    Remember to modify $d011 as needed.

```
        lda #Raster                //Value for $d012 in A
        jsr Sparkle_RestoreIRQ     //Re-enable fallback IRQ
```

- Sparkle_Save (JSR $302)

    Hi-score file saver (A=$01-$0f – high byte of file size, A=#$00 – aborts without saving)

    Sparkle is able to overwrite a single "hi-score file" on the last track of the disk. A default
    hi-score file must be specified in the loader script and Sparkle will use the load address
    of this file during file saving. The saver code and the default hi-score file will occupy the
    last sectors of the last track (track 35 on a standard disk or 40 on an extended disk) with
    a bundle index of $7e and $7f, respectively. These two files can only be accessed by
    bundle index-based loader calls. The saver code requires 2 sectors and the raw hi-score
    file will use 2-16 sectors, allowing a file size of $100–$f00 bytes. Since the last track of
    the disk has only 17 sectors the first block of the saver code may be pushed to the last
    sector of the track before the last (last sector of track 34 or 39).
    To activate the file saving feature, one must first load the saver code using bundle index
    $7e. The drive will first fetch and transfer the first of the 2 saver code blocks containing
    the C64 side of the saver code to the loader buffer of the C64 at $0300–$03ff. 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. The second call in the code
    snippet below will save the number of pages specified in A. The size of the file must be
    multiples of $100 bytes. Sparkle will save from the load address of the original hi-score
    file 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 default hi-score file's size will instruct the loader to return to normal
    loading operations without saving anything. Once saving is complete, Sparkle exits the
    saver loop and returns to loading. To resave the hi-score file, the saver code needs to be
    loaded again. Also, the next loader call must be index-based as we are at the very end of
    the disk.

```
        lda #$7e                //Index of saver code
        jsr Sparkle_LoadA       //Load the saver code to the buffer
        lda #>FileSize          //A=#$01-#$0f, A=#$00 to abort saving
        jsr Sparkle_Save        //Overwrite hi-score file
        lda #$7f
        jsr Sparkle_LoadA       //Reload hi-score file
```

The "Sparkle2\Source" folder includes a Sparkle.inc file with all the important loader labels in Kickass format. This file can be recreated by assembling the SL.asm source file in the same folder. Please use the `-afo` Kickass switch on the command line otherwise the generation of this file may be blocked. <u>Note that assembling the C64 source files is not required for using Sparkle.</u>

REQUESTING A DISK SIDE

Sparkle stores the current disk side's, and the subsequent side'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 found. 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 where the first disk's index is 0. 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.

COMMON ISSUES

Here is a check list to identify common mistakes 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 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. Trying to create a multi-side production from multiple scripts. Multi-side productions must be created from a single script, otherwise the disk sides will not be numbered properly and they may use different Product IDs. You may create separate scripts for each disk side but then you must add them to a single "master script" using the `Script` entry type. Only the first disks info section should contain the `Product ID` and the `ZP` entries.

CAVEATS

While Sparkle can load files compressed by another packer such as Exomizer, make sure to restore the stack pointer and any other registers and zeropage values as required by the packer before you start your program. Restoring the stack pointer will result in overwriting Sparkle's resident code on the stack, so further loader calls will not be possible.

Start the Windows tool from a local or removable drive as it does not seem to work properly from network drives.

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 damages. 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 to spartaofomgATgmailDOTcom.

Please find the most up-to-date version of Sparkle on GitHub: https://github.com/spartaomg/Sparkle2

Sparta, 2019-2022

APPENDIX

VERSION HISTORY

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, Raistlin/G*P, Visage/Lethargy and KAL_123/Miami Fun Project for testing, Rico/Pretzel Logic for bug report, Wacek/Arise for suggestions, 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, maximum 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 the 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.