

***Dreamcast  
Ginsu Sampler  
Disk API***



# Table of Contents

<b>1. Overview</b>	<b>GSA-1</b>
Goals	GSA-1
Features	GSA-1
<b>2. Ginsu Component Definitions</b>	<b>GSA-3</b>
APP = Compliant Executable	GSA-3
BOOT = Boot Executable	GSA-3
XFER = Transfer Executable	GSA-3
INI = Configuration File	GSA-3
<b>3. Ginsu Technical Operation</b>	<b>GSA-5</b>
Flow of Control - Startup	GSA-5
Flow of Control - Transfer	GSA-5
Memory Use During Transfer	GSA-5
<b>4. Using Ginsu in Applications</b>	<b>GSA-7</b>
Overview	GSA-7
Ginsu-enabling Step-by-Step	GSA-7
Ensure Proper Shutdown of Application	GSA-7
Add in the Ginsu Library	GSA-7
Add <code>gsInit()</code> , <code>gsExec()</code> , <code>gsExit()</code>	GSA-8
Test Startup via Ginsu	GSA-8
Test Exit via Ginsu	GSA-8
Handle GDFS Path Translation	GSA-9
Test with GDFS Path Translation	GSA-9
Test with Manifest	GSA-10
Handle GDDA Track Translation	GSA-10

**5. API Reference – Basic Functions ..... GSA-11**

gsInit()	GSA-11
gsExec()	GSA-12
gsExit()	GSA-13
gsExitTo()	GSA-14
gsGDDALogicalToPhysical()	GSA-15
gsGetBasePath()	GSA-16
gsGetArgC()	GSA-17
gsGetArgV()	GSA-18
gsSetUserData()	GSA-19
gsGetUserData()	GSA-20
gsIsActive()	GSA-21
gs_FsOpenEx()	GSA-22

**6. API Reference – Extended Functions ..... GSA-23**

gsSetDefaultExecutable()	GSA-23
gsSetXFerExecutable()	GSA-24
gsSetBasePath()	GSA-25
gsSetFirstGDDA()	GSA-26
gsIniOpen()	GSA-27
gsIniOpenFrom()	GSA-28
gsIniClose()	GSA-29
gsFindApp()	GSA-30
gsGetAppN()	GSA-31
gsGetKeyValue()	GSA-32

**7. INI Files ..... GSA-33**

Application Blocks	GSA-33
Key-Value Pairs	GSA-33
Comments	GSA-33
Sample INI File	GSA-34

**8. Notes: ..... GSA-35**

Not performance oriented	GSA-35
Ginsu Library can be present with no ill effects	GSA-35
Hard Limits of 64/256	GSA-35
Machine State	GSA-35
Memory Scanning	GSA-36
Possible memory overwrite during transfer	GSA-36
Command Line Parser	GSA-36

# **1. Overview**

---

The *Ginsu* library provides an application-transfer system for the Dreamcast. It is intended for use on multi-title GD-ROMs (samplers) built for Shinobi or other similarly lightweight operating systems.

## **1.1 Goals**

- To provide a standard switching mechanism for applications on the same GD-ROM;
- Require minimal resources to limit impact on compliant applications;
- Not impose undue constraints on compliant applications; and;
- Low maintenance requirements.

## **1.2 Features**

- Straight-forward, easy-to-use API
- Small memory footprint (less than 16k)
- Functions to aid in GD-ROM sharing
- “Command-line” parameter passing
- .INI-file based configuration



## ***2. Ginsu Component Definitions***

---

### **2.1 APP = Compliant Executable**

These are the applications that are capable of transfer to and from other applications.

### **2.2 BOOT = Boot Executable**

A `1ST_READ.BIN` that initializes the *Ginsu* environment. It is a compliant APP.

### **2.3 XFER = Transfer Executable**

A small executable program (`GINXFER.BIN` by default) that aids in the transfer between programs by manipulating memory space and machine state. Note that it is not a compliant APP.

### **2.4 INI = Configuration File**

A text file used to describe the components on a GD-ROM. It is easily readable by *Ginsu* compliant applications, but usually not necessary.





## ***3. Ginsu Technical Operation***

---

### **3.1 Flow of Control - Startup**

- Dreamcast launches the BOOT executable
- BOOT executable reads the INI file and sets up the *Ginsu* environment
- BOOT transfers control to *Ginsu*

### **3.2 Flow of Control - Transfer**

- *Ginsu* loads the next APP and the XFER executable into high memory
- *Ginsu* passes parameters to XFER data space
- *Ginsu* transfers control to XFER
- XFER relocates APP to proper run location in low memory (0x8C008000)
- XFER passes parameters to APP data space
- XFER transfers control to the APP
- When APP is done, APP transfer control to *Ginsu*
- (Repeat)

### **3.3 Memory Use During Transfer**

*Ginsu* handles the transfer using high memory as a temporary storage area. Two buffers are used: one for the XFER program (at 0x8cfd0000) and one for the APP (based on file size). The buffer locations are calculated upon demand and are 64k aligned.



## ***4. Using Ginsu in Applications***

---

### **4.1 Overview**

Adding *Ginsu* support to an application is relatively simple. It involves the addition of a few function calls, and paying attention to some basic details.

### **4.2 *Ginsu*-enabling Step-by-Step**

#### **4.2.1 Ensure Proper Shutdown of Application**

The application must have an appropriate exit condition that allows it to exit gracefully – This involves calling the appropriate shutdown, close, exit, etc. calls to disengage the operating system and return the machine to the entry state. For example:

- `sbExitSystem()`
- `njExitTexture()`
- `gdFsFinish()`
- `syMallocFinish()`
- et. al.

One method of testing this is to place a loop in `main()` that re-runs the program after shutdown. Please note that this is not foolproof since static data will be initialized differently!

---

**Note:** This is the most likely location for errors to occur, so it pays to spend some time and verify this.

---

#### **4.2.2 Add in the *Ginsu* Library**

- Add `#include "ginsu.h"` as appropriate to the project
- Add `ginsu.lib` to the linker context

### **4.2.3 Add `gsInit()`, `gsExec()`, `gsExit()`**

These are the three principal functions that *Ginsu*—enable an application. `gsInit()` should be placed prior to other initialization; `gsExec()` goes in the main loop, `gsExit()` is the very last function call of the program.

Note that `gsExec()` is of very little overhead and will not impact overall application performance in any meaningful manner.

---

**Important:** The return values of `gsExec()` should be tested for `GS_FORCE_APP_EXIT` – the application should terminate safely and as soon as possible.

---

### **4.2.4 Test Startup via *Ginsu***

Create a GD-ROM image of the application, with the following changes:

- The supplied `1ST_READ.BIN` (or `GINBOOT.BIN`) should be the BOOT program on the GD-ROM.
- Rename the existing `1ST_READ.BIN` to another name, such as `MYPROG.BIN`
- Add the supplied `GINXFER.BIN` to the image
- Make and add to the GD-ROM image a `GINSU.INI` file of the following form:

```
[GINSU]
Ginsu.FirstApp      = MYPROGRAM
Ginsu.DefaultApp    = MYPROGRAM
Ginsu.XferExecutable = \GINXFER.BIN
```

```
[MYPROGRAM]
Ginsu.Path          = \
Ginsu.Command       = \MYPROG.BIN
```

- Boot the GD-ROM image. It should launch the application successfully.

### **4.2.5 Test Exit via *Ginsu***

With the same configuration, triggering the exit condition should exit the program via *Ginsu*. The result should be the reloading and restarting of the application from the beginning.

## 4.2.6 Handle GDFS Path Translation

To prevent collisions between filenames, titles may be moved into a subdirectory. As a result, applications that load files must be aware of this and handle the situation accordingly.

---

**Note:** Applications using GDFS versions above 1.05 will have the GDFS path translation handled automatically. Applications using older versions of the libraries will have to handle path translation manually as described below.

---

- `gsGetBasePath()` returns a fully-qualified path describing the working directory where the application and data files reside.
- Applications that don't load files can ignore this section
- Application that load all files from the root will need to add a `gdFsChangeDir(gsGetBasePath())` after the file system is initialized and before loading files.
- Applications that use relative navigation via `gdFsChangeDir()` will need to perform an initial `gdFsChangeDir(gsGetBasePath())` after the file system is initialized and before loading files.
- Applications that use absolute navigation via `gdFsChangeDir()` will need to perform a `gdFsChangeDir(gsGetBasePath())` before each of its calls, which must be converted to be relative to the base path.

```
gdFsChangeDir (\\MYDATA)

// becomes...

gdFsChangeDir (gsGetBasePath());
gdFsChangeDir ("MYDATA");
```

Applications that use other methods of file navigation will need to construct proper pathnames based on `gsGetBasePath()`. The return value will be of the form `\\MYDIR\\MYSUBDIR`, it will always contain at least on backslash; it will only end in a backslash if it is the root directory.

## 4.2.7 Test with GDFS Path Translation

- Make the necessary changes in the software to support the path requirements
- Move all the materials into a subdirectory on the GD-ROM image.
- Update the `GINSU.INI` file used previously to identify the directory.

```
[GINSU]
Ginsu.FirstApp      = MYPROGRAM
Ginsu.DefaultApp    = MYPROGRAM
Ginsu.XferExecutable = \\GINXFER.BIN

[MYPROGRAM]
Ginsu.Path          = MYPATH
Ginsu.Command       = \\MYPATH\\MYPROG.BIN
```

- Boot the GD-ROM image. It should launch the application successfully and the application should be able to find it's files.

### **4.2.8 Test with Manifest**

Supplied with the developer materials is a program called `MANIFEST.BIN` – it reads `GINSU.INI` and builds a simple menu to test the Ginsu environment.

- Build a GD-ROM image with the supplied `GD_ROOT` files
- Boot the GD-ROM. You should have a simple menu that allows you to select from modified versions of the Teapot, Tileclip, and F40 demos. ("A" exits the tileclip demo; "B", highlight "Exit", "A" exits the others)
- Add a block of the following form to the supplied `GINSU.INI` and boot the GD-ROM. Your application should be available and functional via the menu.

```
[MYPROG]
Ginsu.Path          = \MYDIR
Ginsu.Command       = \MYDIR\MYEXEC.BIN
Ginsu.FirstGDDA     = 4
Manifest.Name       = "My Program"
```

### **4.2.9 Handle GDDA Track Translation**

CD-ROM and GD-ROM applications refer to data tracks by absolute numbers. A sampler GD-ROM would have multiple applications vying for the same physical tracks, thus they must be translated.

---

**Note:** Applications using GDFS versions above 1.05 will have the GDDA track translation handled automatically. Applications using older versions of the will have to handle track translation manually via `gsGDDALogicalToPhysical()`.

---

# 5. API Reference - Basic Functions

The basic functions are use to support the application transfer and some basic access at parameters.

## 5.1 gsInit()

### Function

GS\_STATUS gsInit (void)

### Input

N/A

### Return Value

GS\_OK                      no errors

### Operation

This function initializes the internal *Ginsu* operation. At this time, it has no effect.

### Usage Example

```
void main (void)
{
    gsInit();
    njUserInit();
    /* ... */
    njUserExit();
    gsExit();
}
```

# 5.2 gsExec ( )

### Function

GS\_STATUS gsExec (void)

### Input

N/A

### Return Value

GS_OK	no errors
GS_FORCE_APP_EXIT	application should exit gracefully, ASAP

### Operation

Currently this function is not used. It is intended for future use to supply heartbeat-related functions, such as a timeout for a POP kiosk with sequential games.

### Usage Example

```
void main (void)
{
    /* ... */
    while (1)
    {
        /* ... main loop operations ... */
        if (gsExec()==GS_FORCE_APP_EXIT)
            break;
    }
}
```



## 5.3 gsExit ( )

### Function

GS\_STATUS gsExit (void)

### Input

N/A

### Return Value

See gsExitTo ( ) for details

### Operation

This function calls gsExitTo ( ) with the default executable name as specified in the GINSU . INI file.

### Usage Example

See gsExitTo ( ) for details

## 5.4 gsExitTo( )

### Function

GS\_STATUS gsExitTo (const char\* cmdline, const char\* path, int firstgdda)

### Input

cmdline	“Command line” for the next application
path	Path for the next application
firstgdda	First gdda track for the next application

### Return Value

This function should never return. In the event of failure, it attempts to reboot the system. If that fails, it will produce an error value:

GS\_NO\_LOAD\_XFER

GS\_NO\_LEAD\_NEXT

GS\_BAD\_XFER\_HEADER

### Operation

Transfer of control to the next application:

- The command line is parsed to identify the next program and parameters – the parsing separates whitespace delimited strings and quoted strings. The parsing is not bullet proof, but is reasonable. The processed command-line is limited to 256 characters; additional arguments are ignored. The next application can retrieve these with `gsGetArgC( )` and `gsGetArgV( )`.
- The file system is initialized
- The XFER program is loaded into high memory. If the program specified by `gsSetXferExecutable( )` is unavailable, it then tries to explicitly load `GINXFER.BIN`. If that too is unavailable, the function fails and exits.
- The next program is loaded into high memory. If the program specified in the `cmdline` is unavailable, it will then try the default program, then `MANIFEST.BIN`, then `1ST_READ.BIN` – if all loads fail, the function fails.
- and the XFER program are loaded into high memory
- The XFER program is verified to be valid
- The `GDDAOffset`, default app, transfer program are carried forward to the next program
- The command line data is passed to the XFER program
- Control is passed to XFER

**Usage Example**

```
void main (void)
{
    gsInit();
    njUserInit();
    /* ... */
    njUserExit();
    gsExitTo("VIEWFILE.BIN MYFILE.EXT", "DATAPATH", 16);
}
```

## 5.5 gsGDDALogicalToPhysical( )

**Function**

```
int gsGDDALogicalToPhysical (int logical)
```

**Input**

Logical	A GDDA track number that would be referenced if this were the only application on the GD-ROM.
---------	---

**Return Value**

A potentially adjusted track number that indicates the correct physical track number to reference instead of the logical.

**Operation**

The given track number is converted to a simple index: Adjusted by the *Ginsu* environment and reconverted to an audio track index.

**Usage Example**

```
int physical_track = gsGDDALogicalToPhysical(4);
gdFsDaPlay (physical_track, physical_track, 0);
```

## 5.6 gsGetBasePath( )

### Function

```
const char* gsGetBasePath (void)
```

### Input

N/A

### Return Value

A pointer to the base path that the application is located in.

The return value will be of the form \MYDIR\MYSUBDIR.

### Operation

The function returns the appropriate path.

### Usage Example

```
gdFsChangeDir (gsGetBasePath());  
gdFsOpen (...);
```

## 5.7 gsGetArgC ( )

### Function

```
int gsGetArgC (void)
```

### Input

N/A

### Return Value

The number of arguments in the command-line buffer. This is always at least 1 since the current application executable name is the first parameter.

### Operation

This function scans the command-line buffer counting the number of arguments.

### Usage Example

```
for (i=0; i<gsGetArgC(); i++)  
{  
    ProcessArg (gsGetArgV(i));  
}
```

## 5.8 gsGetArgV( )

### Function

```
const char* gsGetArgV (int index)
```

### Input

Index of the argument desired.

### Return Value

Pointer to the string that is the specified argument. Null is returned if the argument does not exist.

### Operation

This function scans the command-line buffer looking for the specified argument.

### Usage Example

```
for (i=0; i<gsGetArgC(); i++)  
{  
    ProcessArg (gsGetArgV(i));  
}
```

## 5.9 gsSetUserData( )

### Function

```
GS_STATUS gsSetUserData (const void* src, int size)
```

### Input

Pointer and size of the data block.

### Return Value

GS\_OK

GS\_USER\_PARTIAL\_COPY                      data was too large and was truncated

### Operation

This function copies the first 256 bytes from the source buffer into the *Ginsu* command block. It will be available to the next application via `gsGetUserData( )`

### Usage Example

```
gsSetUserData (myPersistentData, sizeof(myPersistentData));
```

## 5.10 gsGetUserData ( )

### Function

```
GS_STATUS gsGetUserData (void* dest, int size)
```

### Input

Pointer and size of the data block.

### Return Value

GS\_OK

GS\_USER\_PARTIAL\_COPY                      data was too large and was truncated

### Operation

This function copies the first 256 bytes from the *Ginsu* command block into the destination buffer. The data contained within it will be from the last application that called `gsSetUserData ( )`.

### Usage Example

```
gsGetUserData (myPersistentData, sizeof(myPersistentData));
```



## 5.11 gsIsActive()

### Function

```
int gsIsActive (void)
```

### Input

N/A.

### Return Value

Non-zero if *Ginsu* environment is present and initialized.

### Operation

This function allows an application to determine if it is sharing the GD-ROM and should exit or not.

### Usage Example

```
while (!gsIsActive())
{
    play_game();
}

/* shutdown and gracefully exit to next app */
```

## 5.12 gs\_FsOpenEx ( )

### Function

GDFS gs\_FsOpenEx (const char\* fullname)

### Input

Full pathname of desired file to open.

### Return Value

File handle of opened file. NULL if unable to open.

### Operation

This function scans the filename specified and loads the file. It does have side effects – it can and will change the current directory!

- If backslashes are present in the filename, gdFsChangeDir ( ) will be called and the directory state left there.

### Usage Example

```
gs_FsOpenEx (\\MYPATH\\MYFILE.EXT)
```

## 6. API Reference - Extended Functions

### 6.1 gsSetDefaultExecutable()

#### Function

```
GS_STATUS gsSetDefaultExecutable (const char* exename, const char* pathname)
```

#### Input

Name of the executable file that is the default application. This application will be loaded and executed when the `gsExit()` call is made. This function is intended for use by the *Ginsu* boot program.

Pathname of the working directory for the default application. This will be passed onto the application when it is called by the default mechanism.

#### Return Value

GS\_OK

GS\_FILENAME\_TOO\_LONG the limit is 64 characters including terminating null

GS\_PATHNAME\_TOO\_LONG the limit is 64 characters including terminating null

#### Operation

The string is copied to the internal parameter block.

#### Usage Example

```
GsSetDefaultExecutable ("\\SUBDIR\\MAINMENU.BIN");
```

## 6.2 gsSetXFerExecutable()

### Function

```
GS_STATUS gsSetXFerExecutable (const char* exename)
```

### Input

Name of the executable file that is the transfer program. This program will be loaded and executed when the `gsExit()` or `gsExitTo()` call is made. This function is intended for use by the *Ginsu* boot program.

### Return Value

GS\_OK

GS\_FILENAME\_TOO\_LONG the limit is 64 characters including terminating null

### Operation

The string is copied to the internal parameter block.

### Usage Example

```
GsSetXFerExecutable ("\\GINXFER.BIN");
```

## 6.3 gsSetBasePath( )

### Function

GS\_STATUS gsSetBasePath (const char\* dirname)

### Input

Pathname where the next application will be based in. This value will be available immediately by `gsGetBasePath( )` and to subsequent applications. This function is intended for use by the *Ginsu* menu programs.

### Return Value

GS\_OK

GS\_PATHNAME\_TOO\_LONG the limit is 64 characters including terminating null

### Operation

The string is copied to the internal parameter block.

### Usage Example

```
GsSetBasePath ( "\\EXECDIR" );
```

## 6.4 gsSetFirstGDDA ( )

### Function

GS\_STATUS gsSetFirstGDDA (int index)

### Input

Track index where the next application's GDDA tracks will start. This value will be available immediately by `gsGetBasePath ( )` and to subsequent applications. This function is intended for use by the *Ginsu* menu programs.

### Return Value

GS\_OK

### Operation

The value is copied to the internal parameter block.

### Usage Example

```
GsSetFirstGDDA (2);
```

## 6.5 gsIniOpen ( )

### Function

GS\_STATUS gsIniOpen (void\* mem256k)

### Input

mem256k                      A buffer of size GS\_INI\_BUFFER\_SIZE, that is used to manage the INI file.

This memory buffer is needed for all gsIni functions and must be maintained until the gsIniClose ( ) call is made.

### Return Value

See gsIniOpenFrom ( ) for details.

### Operation

This function calls gsIniOpenFrom with the default INI file name, GINSU.INI.

### Usage Example

```
void* inibuf = malloc (GS_INI_BUFFER_SIZE);
gsIniOpen (inibuf);
        /* ... ini processing ... */
gsIniClose();
free (inibuf);
```

## 6.6 gsIniOpenFrom( )

### Function

GS\_STATUS gsIniOpenFrom (void\* mem256k, const char\* filename)

### Input

mem256k	A buffer of size GS_INI_BUFFER_SIZE, that is used to manage the INI file.  This memory buffer is needed for all gsIni functions and must be maintained until the gsIniClose( ) call is made.
filename	The filename that is the INI file to use.

### Return Value

GS\_OK  
GS\_INI\_ALREADY\_OPEN  
GS\_NO\_LOAD\_INI  
GS\_INI\_TOO\_BIG  
GS\_INI\_NOT\_OPEN  
GS\_INI\_SYNTAX\_ERROR  
GS\_INI\_SEMANTIC\_ERROR

### Operation

The function loads the specified INI file (it also tries GINSU.INI on failure) and parses it. The parsing accomodates whitespace delimited strings, square bracketted strings, quoted strings, and semi-colon line comments.

### Usage Example

```
void* inibuf = malloc (GS_INI_BUFFER_SIZE);
gsIniOpenFrom (inibuf, "TEST.INI");
    /* ... ini processing ... */
gsIniClose();
free (inibuf);
```



## 6.7 gsIniClose()

### Function

GS\_STATUS gsIniClose (void)

### Input

N/A

### Return Value

GS\_OK

GS\_INI\_NOT\_OPEN

### Operation

This file closes processing of the INI file opened in gsIniOpen() or gsIniOpenFrom()

### Usage Example

```
void* inibuf = malloc (GS_INI_BUFFER_SIZE);
gsIniOpen (inibuf);
    /* ... ini processing ... */
gsIniClose();
free (inibuf);
```

## 6.8 gsFindApp( )

### Function

GS\_APP\_ID gsFindApp (const char\* appname)

### Input

appname	The application name to whose INI data wants to be accessed.
---------	--

The application names appear in bracketed text in the INI files. Note that these names are not the same as executable filenames. See INI documentation below for more details.

### Return Value

NULL	if the application was not found, or no INI file is open.
	An application unique non-NULL value if it was found.

### Operation

This call must be made between gsIniOpen( ) and gsIniClose( ) calls.

### Usage Example

```
void* inibuf = malloc (GS_INI_BUFFER_SIZE);
gsIniOpen (inibuf);
    GS_APP_ID myID = gsFindApp("TEAPOT");
    if (myID)
    {
        /* ... ini processing ... */
    }
gsIniClose();
free (inibuf);
```

## 6.9 gsGetAppN( )

### Function

GS\_APP\_ID gsGetAppN (int index)

### Input

Index of application to be found.

### Return Value

NULL if the application doesn't exist or if no INI file is open.

### Operation

This function allows the enumeration of INI file applications.

This call must be made between gsIniOpen( ) and gsIniClose( ) calls.

### Usage Example

```
int i = 0;
GS_APP_ID myID;
while (myID = gsGetAppN(i))
{
    /* ... handle this app ... */
}
```

## 6.10 gsGetKeyValue( )

### Function

```
const char* gsGetKeyValue (GS_APP_ID appID, const char* key)
```

### Input

appID	the application ID as returned by gsFindApp( ) or gsGetAppN( )
key	the name of the key whose value is desired.

### Return Value

A pointer to the string if found, NULL if not found.

### Operation

In the INI file, each application has a number of key-value pairs. This function retrieves the desired data if available.

This call must be made between gsIniOpen( ) and gsIniClose( ) calls.

### Usage Example

```
void* inibuf = malloc (GS_INI_BUFFER_SIZE);
gsIniOpen (inibuf);
    GS_APP_ID myID = gsFindApp( "TEAPOT" );
    if (myID)
    {
        const char* str = gsGetKeyValue (myID, "Manifest.Name");
        /* ... more ini processing ... */
    }
gsIniClose();
free (inibuf);
```

## **7. INI Files**

---

The grammar for INI files is very simple. There are a number of application blocks, each of which may contain key-value pairs.

### **7.1 Application Blocks**

An application block is defined by an application name in square brackets. An application name is not the same as an executable filename, since multiple applications may use the same name for files (eg: `START.BIN`).

### **7.2 Key-Value Pairs**

All data is retrieved from the INI file as key-value pairs where both the key and value are strings. The general format is:

Key            =            value

Both keys and values may be in quoted text if whitespace is desired. Quoted text may not span newline.

The convention is to use a two part key name separated by a dot, with the first part indicating the program interested in the value, and the second part being the variable name. See the example below.

### **7.3 Comments**

A semi-colon that is not inside of quoted text denotes a line comment to the end of that line.

### **7.4 Sample INI File**

```
; This is the test INI used to build the Ginsu Library
[GINSU]
```

```
Ginsu.FirstApp      = MANIFEST
Ginsu.DefaultApp    = MANIFEST
Ginsu.XferExecutable = \GINXFER.BIN
```

```
[MANIFEST]
```

```
Ginsu.Path          = MANIFEST
Ginsu.Command        = \MANIFEST\MANIFEST.BIN
```

```
[F40]
```

```
Ginsu.Path          = F40
Ginsu.Command        = \F40\F40.BIN
Ginsu.FirstGDDA      = 4
Manifest.Name        = "F40 Sample Program"
```

```
[TEAPOT]
```

```
Ginsu.Path          = TEAPOT
Ginsu.Command        = \TEAPOT\TEAPOT.BIN
Ginsu.FirstGDDA      = 5
Manifest.Name        = "Teapot Sample Program"
```

```
[TILECLIP]
```

```
Ginsu.Path          = TILECLIP
Ginsu.Command        = \TILECLIP\TILECLIP.BIN
Ginsu.FirstGDDA      = 6
Manifest.Name        = "Tileclip Sample Program"
```

## **8. Notes:**

---

### **8.1 Not performance oriented**

The `Ginsu` INI and command-line functions are feature-oriented and not performance-oriented library. That said, the function calls could probably sit in the middle of the main loop with little or no detriment.

### **8.2 Ginsu Library can be present with no ill effects**

*Ginsu* is self-contained and should not affect the operation of single-title GD-ROMs if present.

### **8.3 Hard Limits of 64/256**

Due to the nature of the task, some small fixed blocks of memory are needed to maintain parameters. As a result, fixed limits do exist for some parameters:

- 64 bytes for executable program names and working directories
- 256 bytes for parsed command line parameters

### **8.4 Machine State**

The SH4 machine state upon entry to a new program is as follows:

- MACH/MACL = 0
- VBR = 0x8c00f400
- GBR = 0x8c000000
- PC/SPC = 0xac010000
- SR/SSR = 0x700000f0
- PR = 0

All other elements are as left by the exiting program

## 8.5 Memory Scanning

The XFER program scans the application it's about to run looking for parameter blocks. It is unlikely that a conflict would exist, though it is remotely possible. The scan logic looks for 4 contiguous longwords with the following properties:

- 'GINS' header, which appears as 'SNIG' due to endianness
- >1 – a version number
- 0x3ac59e76, a magic number chosen from thin air
- >0x200 – the size of the parameter block

## 8.6 Possible memory overwrite during transfer

The transfer process needs to have the *Ginsu* library, the next application, and the transfer program in memory at once. Most of the process is fairly safe from problems, however one potential point of failure does exist. If the *Ginsu* library is high in memory, then the target program or transfer program could be loaded on top of it.

This is extremely unlikely to occur in most cases, but if the programs involved are extremely large (combined size > 15MB), or run in high memory, this can be an issue.

## 8.7 Command Line Parser

The command line parser is a simple parser of arguments. It is reasonably thorough. Normally:

- Whitespace delimits arguments
- Quotes (") delimit arguments containing whitespace; A pair ("" ) resolves to one (")
- Whitespace is trimmed from non-quoted arguments
- When quoted arguments are read via `gsGetArgV( )`, they contain only the text within the quotes, not the quotes themselves.

If the data fed to the parser is aggressive, some slight anomalies will occur:

- "TOKEN"NEXT – with no spaces will yield two arguments of TOKEN and NEXT
- "" – with no spaces will cause early termination of the argument list