GVARIANT: EFFICIENT PARTIAL DESERIALISATION

GVARIANT: EFFICIENT PARTIAL DESERIALISATION

By
RYAN LORTIE

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE (2008)
(Computer Science)

McMaster University
Hamilton, Ontario

TITLE:	GVariant: Efficient Partial Deserialisation
AUTHOR:	Ryan Lortie
SUPERVISORS:	Alan Wassyng
	Wolfram Kahl
PAGE COUNT:	viii, 225

Abstract

This work documents the creation of a new serialisation format. Developed for use in the GNOME platform, the requirements for this serialisation format are based on the unique needs of the community, plus some "guiding principles" that have developed in the community over the years.

The serialisation format is particularly designed to allow for rapid deserialisation — which is expected to be the most common use case — with most operations occurring in a small constant time (regardless of the size of the data).

Finally, a complete implementation of the serialisation format — called GVariant — is presented. GVariant models each value as an object with an API that is both convenient for GNOME programmers and has a flavour that they are familiar with.

Acknowledgements

My sincere thanks go to my supervisors, Dr. Alan Wassyng and Dr. Wolfram Kahl, for all the support and guidance they provided and particularly for their graceful handling of the consequences of my tendency to procrastinate.

This Document

CC 3.0 +BY +SA

You are free:

to Share
to copy, distribute and transmit this document

to Remix
to modify, excerpt or adapt the work

Under the following conditions[1]:

Attribution.
You must attribute me in a reasonable manner (but not in any way that suggests that I endorse you or your use of this document).

Share Alike.
If you alter, transform, or build upon this document, you may distribute the resulting work only under the same or similar licence to this one.

This notice has no limiting affect on your existing fair dealing rights.

All Executable Code

GPL 2.0, LGPL 2.1, (L)GPL 3.0+

GVariant is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the licence, or (at your option) any later version.[2]

GVariant is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

[1] detailed licence terms are available at `http://creativecommons.org/licenses/by-sa/3.0/ca/`

[2] the full text of the GNU Lesser General Public License is available at http://www.gnu.org/copyleft/lgpl.html or by writing to The Free Software Foundation

Table of Contents

Preface

Introduction

This thesis details the design and development of a high performance serialisation system for use within the GNOME desktop. The serialisation system is primarily developed for use in a configuration settings system which is also targeted at the GNOME community. The work is, fundamentally, divided into three parts.

The first part is a requirements gathering phase. There are a number of common "folk knowledge" principles of good programming that have gained wide acceptance within the GNOME community over the years but have never been gathered in to one place. The principles, as they apply to the work here, are documented. These principles, in a large part, form the definition of "high performance" as used in the previous paragraph.

Additionally, as with any community, there is a wealth of background information about the GNOME community and the tools and libraries that its members are accustomed to that will strongly affect any system designed to be used within it. This information is also documented.

These two pieces of information are used to motivate the design of the configuration settings system and from there, the requirements placed on its value serialisation system.

The second part involves the specification of a newly developed serialisation format. The primary design criteria of this serialisation format is that it can be implemented by a program that adheres to the principles presented in the first part. Additionally, this implementation of the format must have the property that it "fits in nicely" with the expectations of the GNOME community, as detailed in the background information.

The third part presents a new implementation of the serialisation format, as specified, presented as a new library. This library adheres to the principles outlined in the first part, and perhaps more importantly, allows for the development of programs that also adhere to these principles. This acts as a demonstration that the design criteria of the serialisation format have been met.

PART I

This part contains background information required to understand both the motivation behind the creation of GVariant and the design decisions made as a result of the community by which GVariant will be used. From this background information, formal requirements are developed and specified.

Chapter 1

Background

The development of GVariant is in direct response to a need encountered while addressing deficiencies in the current GNOME Desktop platform.

With this in mind, this chapter presents background information about the GNOME community and some projects commonly used in the development of GNOME applications that are relevant to this work.

1.1 The GNOME Desktop

The GNOME project (see [**GNOME**]) is a Free Software desktop environment with the goal of providing an easy-to-use graphical interface for normal people, and a rich set of development libraries for programmers.

GVariant fits into the second role. The main influence on the design of GVariant is that it should be a joy for GNOME application developers to use. For this reason, many of the design requirements directly relate to community expectations. The following information led to the "Context Requirements" outlined in Section 4.2:

- GNOME is programmed primarily in the C programming language. The foundation libraries are written entirely in C.

- GNOME developers are very familiar with object-oriented programming using the GObject object system. This object system contains many conveniences to overcome limitations of the C programming language when used to write object-oriented programs.

- The GObject type system includes a generic value type, GValue. GValue is designed to contain dynamic program state — not to transmit or store persistent data. It stores, for example, native C types (which may be different sizes on different platforms) and pointers (which are not easily transmitted in a meaningful way).

- The DBus message bus enjoys wide popularity among GNOME developers and has been very successful in the free desktop world in general. It has been designed to allow cross-process and cross-host communication in a platform-agnostic way.

- In addition to traditional desktop roles, GNOME is also used on embedded and mobile computing platforms where the memory and processing resources are often very limited.

1.2 GConf

Currently, the GNOME desktop uses the GConf project (see [**GConf**]) for storage of application preferences and settings. GConf — despite having introduced some novel features that are widely appreciated — is regarded within the community as having some serious design and performance issues.

Among the fundamental problems with GConf, it is particularly worth mentioning these two in terms of motivating the content of this work:

- GConf uses a value system called GConfValue. GConfValue is rather limited with respect to the types of values that it can express.

Additionally, aside from GConf, GConfValue is used for nothing else in the desktop. This means that conversion is always required before storing settings in the configuration database.

GConfValue is capable of expressing strings, integers, floats, and booleans. It is also capable of expressing pairs or lists of these things, but not in a typesafe way (eg: a list of strings has the same type as a list of integers) or in a way that supports recursion (eg: you can not have a list of pairs). The limitations of the expressiveness of GConfValue have caused some users to resort to hacks like storing an XML blob as a string in GConf.

- GConf uses a client/server-based architecture wherein all queries for settings made by application programs are handled by a single server process. During login, when many applications are starting at the same time this leads to a flurry of context switches and a substantial amount of serialisation (in a process which ought to be significantly parallel).

These issues, along with others, have been identified as being so fundamental that to correct them is to practically require the replacement of GConf. The desire to do so is what led, indirectly, to the work described in this thesis.

1.3 DBus

DBus is a message bus system specified in [**DBus**]. It is the most commonly-used mechanism for interprocess communication on the GNOME Desktop.

The name "DBus", strictly speaking, refers to the specification of the message bus protocol. In theory, many implementations of this protocol could exist.

Several independent client side library implementations exist, written in high level languages. In practice, however, the name "DBus" has become synonymous with the reference implementation of the client

side library and bus daemon that were developed in C, in parallel with the specification. More specifically, these two pieces of software are called `dbus-daemon` and `libdbus`.

As mentioned, DBus enjoys wide popularity among GNOME developers; many programs in the GNOME desktop link against `libdbus` and make use of its APIs. GNOME developers are familiar with DBus concepts, and are particular familiar with its type system.

One limitation of the DBus API is that the smallest object that can be dealt with is a message. A message may contain several arguments, but the arguments can not be treated as individual objects. An iterator interface is used to construct or deconstruct the message, all at once, from base C types (integers, strings, etc.).

Chapter 2

Community Folk Knowledge

This chapter outlines a number of "best common practice" principles that have become popular knowledge in the GNOME community over the past years. These principles are not documented anywhere[1], but most of them will be familiar to anyone who has been around long enough.

These principles are used as guiding principles because they are believed to be sound. The author has personally witnessed the effects when these principles have been applied to make modifications to modules in the GNOME desktop — better performance, faster startups, lower memory consumption and better battery life. The principles also have a common sense aspect to them; when you think about it, it is clear what should be done. When these principles have not been applied it is most often because the programmer simply didn't think about them.

The GNOME community is the first community to have had a "planet". "Planet GNOME" is a website that aggregates the blog entries of many individual GNOME developers into a single "river of news". This

[1] A "GNOME Goals" project exists (see http://live.gnome.org/GNOMEGoals) to address some common issues with GNOME applications but this project focuses on specific issues (like ensuring that applications are using specific library features) rather than addressing the sorts of problems listed in this chapter.

website is frequented by nearly every active GNOME participant and by thousands of other readers.

Every now and then a member of the community notices a particular programmer practice that is causing some sort of problem (such as poor performance or unnecessary memory consumption) and brings it to the attention of the community by posting an entry to their blog (which is then picked up and published by Planet). In the case that the practice is widely agreed upon to be problematic, a witch-hunt usually ensues. The typical process involves a number of interested individuals searching for instances of the problem in the various modules that form the GNOME desktop. As problems are found, these individuals either write patches to address the issue or guilt the maintainer of that module into doing so.

The GNOME platform libraries are under a strict policy that ensures that API-incompatible changes are not made during a major release series (eg: 2.x). These long-term stability constraints are in place to make the GNOME platform more attractive to application developers who want to avoid having to constantly update their application to be compatible with "the latest version". Occasionally, solving some newly-discovered problems would require making changes to the API of the library in question. Since this is not acceptable, some instances of the problem cannot be solved.

For this reason, when developing a new software library for use in the GNOME platform it is critically important to ensure that best common practices are adhered to during the initial development of the library; it might not be possible to solve problems at a later date.

This list is presented as a list of mistakes that have commonly been made when developing programs for GNOME. For each item, the best common practice approach for avoiding the problem is given.

2.1 Bad File Access Patterns

When Moore's Law still applied to single-core processor development, processors and memory buses gained speed at an exponential rate.

During the same period of time, however, the speed of a disk seek improved very little.

When a graphical application starts, there are typically many small resource files that must be loaded in order for the application to function. These resource files are things like fonts, user interface layout descriptions and icons.

In the case of icons, for example, each icon might be stored in a separate `.png` file. These individual files could be stored in almost any place on the disk. If during its startup an application needs to read 100 icons from the disk then a huge amount of seeking is going to be required.

Within GNOME, this problem has generally been dealt with by creating "cache files". In the case of icons, a file (named `icon-theme.cache`) is created in the root directory of each icon theme. This file contains a copy of every icon in the theme. The file is almost always stored in a contiguous (or nearly contiguous) section of the disk, so icon accesses are localised, reducing the amount of seeking.

2.2 Unnecessary System Calls

Another practice that has been flagged as a problem in the desktop is performing storms of unnecessary system calls. This often results from a situation such as a programmer of a function thinking that a `stat()` system call (to check for the existence of a file) is probably pretty cheap. Taken in isolation, this may be true, but if that function is called a thousand times in a rapid succession then the result is a program senselessly asking the kernel, a thousand times in a row, if a certain file exists.

As another example, consider a naïve implementation of a reader of the icon cache file mentioned in the previous section: each time a new icon is requested, open the cache file, find the icon, read it, then close the file. During startup, when 100 icon are read in a row, this practice of closing and immediately reopening the file, when viewed from the outside, looks extremely silly.

System calls on Linux (which is where GNOME is mainly used) are relatively fast compared to other operating systems. They still require an lot of work, however — even in the case where the information requested is in a warm disk cache. The processor has to save the entire state of the running process, context switch into a higher privilege mode, switch to a new execution stack, and find and execute a system call handler. This is in addition to the actual work that is done (like traversing the directory tree to find a file) and when the call is done the entire process needs to be reversed. This unnecessary computation is wasteful if it can be avoided.

It is considered best practice to keep an eye on the number of system calls that are being issued and to reduce this number where it is possible to do so without making additional sacrifices.

2.3 Unnecessary Per-Process Memory Use

One observation that was made about GNOME some time ago is that when all of the various shared libraries that are used in a typical GNOME application are loaded and initialised, the application is already using a lot of memory before it has even done anything. This memory is used by every process that is running as part of the desktop, so in a desktop with n processes the effect is multiplied by n. Since very little has happened in each process at the time of initialisation the contents of this memory is nearly identical in every single one of these processes. This is clearly very wasteful.

As is typical, a small crusade was launched to solve this problem. The focus at first was to reduce the size of n. Some results came of these efforts, such as a focus on making applications "single instance" (wherein multiple document windows are displayed by a single process) and by the combining of many background services into larger common service daemons.

The problem with this approach, however, is that the memory protection facilities of modern operating systems are designed to limit the amount of damage that can be caused by a wide range of programming errors

to a single process. Increasing the amount of responsibility that each process has also increases the damage that can be caused by a single failure. Google's new Chrome browser (which is Free Software, but not a part of GNOME) makes a point of this issue by running each separate tab of the browser in its own separate process.

As opinion shifts away from the idea that reducing *n* is a good idea the focus shifts to the problem of trying to reduce the per-process memory overhead. This is a fascinating field of work. There are a huge number of contributors to this problem and a full discussion of it would make a very interesting paper, but is far out of context here. One non-obvious example is the overhead caused by the dynamic linker when it copy-on-writes memory pages during relocation of shared libraries.

Another example that has seen considerable work done in the GNOME community are per-process tables and hash tables of information. Two examples from font handling are the hash table of which fonts (by name) are installed and which files they are contained in and the kerning tables used for inter-character spacing when rendering text.

In the case of the font name table, the old approach (as implemented by the `fontconfig`[2] library) was to scan the system font configuration at each startup to get the list of fonts. This information was parsed into a hash table to allow for quick lookup of a font when it was required. The problem caused by this approach is that each application ends up with an identical copy of this hash table in its address space.

Currently, `fontconfig` is an awful lot smarter. A master hash table is created and stored to a file. This file is designed so that it can be memory-mapped into the address space of a process (consult [**ast**] for an introduction). Memory mapping a file allows every process on the system to share the copy of the file that is in the kernel's disk cache. Even though each process is still using this memory it is now using a shared copy of it — the price of having the hash table is amortised across all the processes.

[2] http://www.fontconfig.org/

In general, having these sorts of memory-mapped cache files for common per-process data is an "in" thing in the GNOME community these days. In 2005, an abstract interface for memory-mapped files was added to GLib as a new utility class named `GMappedFile`.

2.4 Unnecessary Faults

As more and more things are pushed into large memory-mapped cache files, particular care has to be taken to ensure that these files are accessed carefully.

Modern operating systems, when asked to memory map the contents of a file, do not normally read that file from disk. Instead, they mark the memory region into which the contents of the file were mapped as an invalid region and wait to receive a page fault from the processor. At this point, the pages are loaded from the disk into memory and made available to the process (and the process is resumed and allowed to read the memory as if nothing happened).

When potentially expensive operations occur implicitly as a result of something as innocuous as a memory read, care needs to be taken by the programmer to ensure that they are not accessing any more memory than is necessary.

Some might warn that this is an implementation problem, and attempting to address it at the design stage is a case of inappropriate micro-optimisation. Consider, though, that it is possible for a file format (which may become set in stone by compatibility requirements) to mandate these bad access patterns. Commonly accessed data may be spread across many pages (with less-commonly used data intermixed). Simple operations may require chasing a number of pointers (again, across several pages).

A number of tools have been developed by the GNOME community to spot exactly these sorts of problems. The most noteworthy of these tools

is `iogrind`[3] which is capable of showing the access patterns of a program down to page-level granularity.

2.5 Excessive Round Trips

Numerous services run as part of the GNOME desktop. An example is the GConf configuration server, which is responsible for providing access to configuration settings to applications. GConf also consists of a client-side library which accepts requests from the programmer and communicates with the server to get the work done.

A simple way of arranging this would be to have the client-side library request a configuration key from the server for each request that is made against the library. In the case that an application is starting up, however, it is probably requesting many settings from the configuration server.

Each request to the configuration server involves a complete context switch to another process (which is considerably more expensive than merely context switching into the kernel). Many processor caches (such as the TLB) must be flushed and others are effectively flushed as a result of executing on a totally different working set.

The problem cannot be avoided in the same ways as avoiding unnecessary system calls since each request is for different information. Since both sides of the interface are part of the same project, however, there is an increase in flexibility. Support for new types of requests can be added to the interface.

The way that GConf has solved this problem is by requesting an entire subtree of the configuration settings database at once, in a batch. These sorts of "batch access" calls are an effective way of reducing the number of round trips.

[3] http://live.gnome.org/iogrind

As implemented in GConf, however, this solution has led to another problem: each client end up with a second copy of the configuration settings in a cache on the client side. Not only does this waste memory, but there is a new cache-coherency issue — the local copy must be invalidated when the server's copy changes. This has led to considerable complication of the design of GConf.

In general, a better way to avoid this problem is to develop interfaces and protocols that completely eliminate the need for round trips. An interesting (and somewhat ancient) example of this can be found in the X11 protocol. Each window on the screen has a unique identifier. After a client creates a new window it is almost certain to immediately perform some operation that depends on knowing that identifier. Instead of allowing the server to allocate identifiers to new windows, a unique range of identifiers is provided to each client so that the client can set the identifier as part of the window creation request (and therefore know what its value is without waiting for a reply from the X server). This allows clients to send whole strings of requests to set up entire window hierarchies at once.

2.6 Excessive Startup Work

In a typical desktop configuration, the resources of a computer are remarkably underutilised most of the time. Even when "using" the computer by browsing a web page or writing a document, the user is using tiny fractions of the computational, memory and IO resources available.

The performance of a computer under these workloads is not judged by how fast the user can accomplish their task, but by how fast the computer responds to their requests. The classical example of this is how fast the computer starts a program. The classical classical example is how fast a computer starts up.

To a very large extent, the performance of a modern operating system is judged by how fast it "boots up" or "logs in". For this reason, a great deal of effort goes toward minimising these intervals.

One of the easiest ways to do this is simple deferral of work. If something is not strictly required at startup time then it should be deferred to a later time so that more important tasks can occur.

As a simple example of this practice, consider the applet on Ubuntu systems that periodically checks for updates and informs you when they are available. When this applet starts, the first thing that it does is to sleep. Only after 30 seconds have passed does it initialise itself. This is done to "make way" for other more important and user-visible processes to start faster.

In general, if work is not absolutely required during the login process then it should not be performed until later.

2.7 Blocking the Mainloop

One of the most visible problems when developing applications for GNOME (or practically any graphical environment) is blocking the mainloop. GNOME applications are implemented using an event-driven mainloop that, in a single thread, monitors a number of different sources of events. When a new event occurs it is dispatched to the handler function that has been registered with the mainloop.

One such class of events is user input events. These events are dispatched to GTK, which is responsible for refreshing the appearance of the UI in response to them.

The problem comes when an event handler takes too long. This could either be because it is performing a lengthly computation or because it is blocking while waiting for data (from another application, the network, the disk, etc). In either case, this work is said to be "blocking the mainloop". The result is that, for the duration of the event handler, the application is unresponsive to additional input events.

Even if the blocking is only for a quarter of a second, it can be enough to make the application feel unresponsive.

There are many common ways to deal with this problem. Asynchronous (non-blocking) IO operations are used where possible. It is also common to place long-running computations into a separate thread so that the mainloop can run at the same time as the computation.

This principle is not directly applicable to the work covered in this document (other than maybe to provide additional motivation for making things fast). Being a particularly common and visible pitfall, it is stated for completeness.

2.8 Unnecessary Wakeups

An unnecessary wakeup occurs when the kernel schedules a process to run and no real work is done. Running the process was unnecessary and time has been wasted.

There are two main situations in which this occurs.

The first case is when an application decides that it should wake up periodically to "poll" some condition (such as a file having been modified) that it needs to take action on. Of course, most of the time the file has not been modified, so the process goes directly back to sleep. This may seem innocent enough, particularly if these probes only occur when the system is otherwise idle, but these wakeups cause serious problems.

Modern processors (particularly the variety used in laptop computers) have the ability to go into low power states when they are inactive. Longer periods of inactivity permit deeper reductions in power consumption. If ten processes on the system are each polling a condition ten times a second then the kernel is causing the CPU to wakeup 100 times per second and the effectiveness of these power saving features is greatly diminished. Your laptop battery doesn't last as long.

The solution to this problem is to find an event-driven interface and use it instead of polling; this results in the process only waking up when it actually needs to do work.

The second case is when an event occurs and a number of processes have registered interest in this event. Each process needs to be notified. In many cases, however, only a very small subset of the notified processes will perform an action in response to the event.

This is often caused by poor granularity in the interface for registering interest in being notified about a particular class of events. As an example, say a process is interested in the name displayed on the title bar of an application window. It wants to be notified when the title changes. The title of the window is stored as a property of the window, so it needs to register for property change notifications with the X server. Unfortunately, this interface does not allow registration of interest in only some properties — it's all or none. The process will now be woken up for every property change, including the "most recent user interaction timestamp" property which is updated with every single keystroke.

This particular problem has recently been worked around by moving the user interaction timestamp property to a separate proxy window that exists specifically so that the updating of this property is not broadcast to all property notification listeners. In general, these problems are best avoided in the first place by providing interfaces that allow for highly granular registration of interest in notification events.

Chapter 3

GSettings

The work described in this thesis — GVariant — is primarily designed for use within the larger project that is GSettings.

For the reasons set out in Section 1.2, GSettings is being developed, by the author, as a replacement for GConf.

In order to better understand some of the decisions that have affected the design of GVariant it is worth having a look at the design decisions that were made for GSettings.

This chapter outlines the basic design of GSettings, making reference to the background information and best principles that have been outlined in the last two chapters.

3.1 High-Level Interface

GSettings is a high-level interface for access to a dictionary of strongly-typed keys. It can have many implementors.

Every implementor, however, is expected to ensure that all access is performed according to a schema. A schema is a finite mapping of key names (strings) to types and default values.

Any attempt to access a setting using a key name not in the schema produces a runtime error. Any attempt to access a key that is specified in the schema, however, will always result in a value of the expected type being returned.

In keeping with the "no unnecessary wakeups" principle of Section 2.8, GSettings features an API to notify the programmer when a key has changed its value.

GSettings, of course, attempts to correct many shortcomings of the design of GConf.

As mentioned in Section 1.2, GConf features a type system that is not used elsewhere in the GNOME Desktop. GConfValue doesn't exist for a current[1] lack of other type systems in the GNOME platform. One goal of GSettings has been to reduce the number of type systems in use.

GSettings tackles this problem by using the type system of DBus. Any value that can be sent over DBus is suitable for storage as a value in the settings database. This type system is powerful and widely understood and used.

GValue was another contender for use, but its ability to contain pointers and its inconsistent representation across multiple platforms prevented this possibility. Additionally, the type system of GValue is geared more towards object-oriented programs rather than describing the format of serialised data.

The first (and at present, only) implementor of the GSettings interface is one which stores settings in a lower-level backend database called `dconf`.

[1] At the time that GConf was created, Neither GValue nor DBus existed, so GConfValue was needed at that time. When GValue and DBus were implemented, GConfValue proved to be too limited to base either of these two systems on.

3.2 dconf

dconf is a very simple hierarchy of keys, with each key having a value. No type-checking is performed at this level; anything goes.

A typical dconf configuration will feature a number of separate backend databases used to enforce various levels of system policy and default settings. Access to each of these backends is performed using a unique approach in order to avoid many of the pitfalls listed in Chapter 2.

All settings are stored in a single file (reducing bad file access patterns as described in Section 2.1). This single file is then memory-mapped into the address space of every process which wants to read settings. For reads, this avoids roundtrips to a settings server (Sections 2.5 and 2.2). It also means that no process ever has to have its own copy of any settings (addressing the problem outlined in Section 2.3).

In fact, when reading a value such as an array of integers from the configuration system, the programmer can obtain a pointer directly to that array, as stored in the memory mapped region. There are no copies made.

Access to the configuration database is lockless and safe for concurrent access by many readers and one writer. For this reason, all write operations must be directed through a settings server. Changing settings is a much less common case than reading settings (which is something that occurs many times every time a program starts). Also, since settings are not changed as part of the login process, the setting server doesn't need to be (and isn't) started at this time (addressing Section 2.6).

This arrangement avoids the problems outlined in the second point of Section 1.2.

The safe concurrent lockless access mixed with the fact that the file is to be memory mapped and accessed directly as native C data implies that the file format used by dconf must have a binary format. It would be extremely awkward to attempt the lockless concurrency and by

definition impossible to store native C data with a text file. This implies that we must have support for serialising values to a binary serialisation format.

3.3 GVariant

As the author was writing `dconf`, it quickly became clear that a large part of the work was involving the design and implementation of a serialisation format that was capable of being implemented in a way that would support all of the goals of `dconf` and not violate the principles outlined in Chapter 2.

For reasons of encouraging loose coupling between the settings system and this significantly complicated subproject, it was decided to split the development of the serialisation format — and the value handling system in general — into its own project. This project is what is documented in this thesis.

During the development of this subproject, the author was approached by several others who were interested in making use of the work for things other than settings storage.

In the interest of encouraging code reuse and allowing even wider use of the DBus type system, the work described here is planned for inclusion in GLib[2] and will thus be available for use by all GNOME applications.

[2] GLib is a general-purpose utility library which provides many useful data types, macros, type conversions, string utilities, file utilities, a main loop abstraction, and so on. This library is the lowest-level library in the GNOME stack and, as such, as part of every GNOME application.

Chapter 4

Requirements

GVariant was created from requirements that led to some unusual design considerations. This section documents these requirements.

The serialisation format, which is described in Part II, has been designed specifically to allow an implementation to conform to the requirements listed here.

4.1 The Need for GVariant

The need for GVariant has arisen from the lack of a system in the GNOME platform for representing and serialising complex strongly-typed data (such as strings, integers, floating point numbers and arrays and tuples of these things). This gap in the platform first became apparent while attempting to write a configuration storage system. This configuration system requires the storage of user-specified data on persistent storage and communication of this data via sockets to a configuration server.

As a result, the primary requirements of GVariant are that it can be used in the following ways:

Usage Requirement 1
Construct and deconstruct complex data values.

Usage Requirement 2
Construct and deconstruct complex data values.

Usage Requirement 3
Save and load these data values to and from disk.

Usage Requirement 4
Send and receive these data values over sockets.

With the original user of GVariant being a configuration system, use of GVariant is expected to be very read intensive; configuration settings are read each time a program is started, but rarely changed. Other planned users of GVariant operate in the same way. This introduces the following guiding principle which is not a hard requirement:

Guideline 1
GVariant must be optimised for reading and deconstructing values. Constructing and writing may be less efficient.

4.2 Context Requirements

Since GVariant will be used most extensively by the GNOME community, the following requirements are made in accordance with the background information provided in Section 1.1:

Context Requirement 1
GVariant must be written in C.

Context Requirement 2
GVariant must be friendly for use by C programmers.

Context Requirement 3
GVariant must allow direct access to the serialised data (by exposing pointers) in cases where the serialised data can be easily interpreted as a native C type (for example, arrays of integers).

Context Requirement 4
GVariant must use an object-based API. Each value must be presented to the programmer as an object. It must be possible to group multiple objects into a container object and to extract child objects back out again.

Context Requirement 5
GVariant must contain conveniences, where possible, to overcome limitations of C. These conveniences should operate in ways that are familiar to those who are familiar with GObject.

Context Requirement 6
GVariant must use the type system of DBus to facilitate the possibility of sending GVariant values over this bus.

4.3 Performance

Several requirements have been made to ensure efficient operation of GVariant.

A holistic approach has been taken in gathering and specifying these requirements. For example, explicit treatment has often been given to the operating system concepts outlined in Chapter 2. As a result of this approach, many requirements that would normally be presented as "non-functional" are functional requirements. Where possible, we speak not of "it must be fast"; rather, slow operations are directly forbidden as a matter of functionality through use of language such as "must not fault in pages from the disk other than when they are absolutely required".

Performance Requirement 1
In keeping with the principle of not using unnecessary memory by having multiple copies of the same data (Section 2.3) GVariant must not make unnecessary copies of data.

Performance Requirement 2
Particularly, GVariant must allow use of shared memory between processes (as described in Section 2.3), including sharing memory with the operating system page cache when reading from disk.

Performance Requirement 3
As per Section 2.4, GVariant must not fault in pages from the disk other than when they are absolutely required. This implies that GVariant must access as few bytes of data as is possible when performing any operation.

Performance Requirement 4
GVariant must perform operations quickly, particularly in its primary use case of reading data from a mapped file. Nearly every deserialisation operation in this use case must occur in constant time.

PART II

This part specifies the new serialisation format, including description of the type system. It also provides notes for those who may be interested in implementing this format, including some observations about how seemingly linear-time operations can be implemented in constant time (along with proof of correctness).

Chapter 5

Types

As per Context Requirement 6, GVariant must be substantially compatible with the DBus message bus system (as specified in [**DBus**]).

To this end, the type system used in GVariant is almost identical to that used by DBus. Some very minimal changes were made, however, in order to provide for a better system while still remaining highly compatible; specifically, every message that can by sent over DBus can be represented as a GVariant.

Some baggage has been carried in from DBus that would not otherwise have been present in the type system if it were designed from scratch. The object path and signature types, for example, are highly DBus-specific and would not be present in a general-purpose type system if it were to be created from scratch.

5.1 Differences from DBus

In order to increase conceptual clarity some limitations have been lifted, allowing calls to "never fail" instead of having to check for these special cases.

- Whereas DBus limits the maximum depth of container type nesting, GVariant makes no such limitations; nesting is supported to arbitrary depths.

- Whereas DBus limits the maximum complexity of its messages by imposing a limitation on the "signature string" to be no more than 255 characters, GVariant makes no such limitation; type strings of arbitrary length are supported, allowing for the creation of values with arbitrarily complex types.

- Whereas DBus allows dictionary entry types to appear only as the element type of an array type, GVariant makes no such limitation; dictionary entry types may exist on their own or as children of any other type constructor.

- Whereas DBus requires structure types to contain at least one child type, GVariant makes no such limitation; the unit type is a perfectly valid type in GVariant.

Some of the limitations of DBus were imposed as security considerations (for example, to bound the depth of recursion that may result from processing a message from an untrusted sender). If GVariant is used in ways that are sensitive to these considerations then programmers should employ checks for these cases upon entry of values into the program from an untrusted source.

Additionally, DBus has no type constructor for expressing the concept of nullability[1]. To this end, the Maybe type constructor (represented by `m` in type strings) has been added.

[1] A "nullable type" is a type that, in addition to containing its normal range of values, also contains a special value outside of that range, called `NULL`, `Nothing`, `None` or similar. In most languages with reference or pointer types, these types are nullable. Some languages have the ability to have nullable versions of any type (for example, "`Maybe Int`" in Haskell and "`int? i;`" in C#).

Some of these changes are under consideration for inclusion into DBus[2].

5.2 Enumeration of Types

5.2.1 The Basic Types

Boolean
A boolean is a value which must be True or False.

Byte
A byte is a value, unsigned by convention, which ranges from 0 to 255.

Integer Types
There are 6 integer types other than byte — signed and unsigned versions of 16, 32 and 64 integers. The signed versions have a range of values consistent with a two's complement representation.

Double Precision Floating Point
A double precision floating point value is precisely defined by IEEE 754.

String
A string is zero or more bytes. Officially, GVariant is encoding-agnostic but the use of UTF-8 is expected and encouraged.

Object Path
A DBus object path, exactly as described in the DBus specification.

[2] Considerable discussion has been made in face-to-face meetings and some discussion has also occurred on the DBus mailing list: http://lists.freedesktop.org/archives/dbus/2007-August/008290.html

Signature String
A DBus signature string, exactly as described in the DBus specification. As this type has been preserved solely for compatibility with DBus, all of the DBus restrictions on the range of values of this type apply (eg: nesting depth and maximum length restrictions).

5.2.2 Container Types

Variant
The variant type is a dependent pair of a type (any of the types described in this chapter, including the variant type itself) and a value of that type. You might use this type to overcome the restriction that all elements of an array must have the same type.

Maybe
The maybe type constructor provides nullability for any other single type. The non-null case is distinguished, such that in the event that multiple maybe type constructors are applied to a type, different levels of null can be detected.

Array
The array type constructor allows the creation of array (or list) types corresponding to the provided element type. Exactly one element type must be provided and all array elements in any instance of the array type must have this element type.

Structure
The structure type constructor allows the creation of structure types corresponding to the provided element types. These "structures" are actually closer to tuples in the sense that their fields are not named, but "structure" is used because that is what the DBus specification calls them.

The structure type constructor is the only type constructor that is variadic — any natural number of types may be given (including zero to form the unit type, and one).

Dictionary entry
The dictionary entry type constructor allows the creation of a special sort of structure which, when used as the element type of an array, implies that the content of the array is a list of key/value pairs. For compatibility with DBus, this binary type constructor requires a basic type as its first argument (which by convention is seen to be the key) but any type is acceptable for the second argument (by convention, the value).

Dictionary entries are as such by convention only; this includes when they are put in an array to form a "dictionary". GVariant imposes no restrictions that might normally be expected of a dictionary (such as key uniqueness). The DBus specification specifies that keys should be unique, but also declares that — for performance reasons — implementations need not enforce this.

5.3 Type Strings

Just as with DBus, a concise string representation is used to express types.

In GVariant, which deals directly with values as first order objects, a type string (by that name) is a string representing a single type.

Contrast this with "signature strings"[3] in DBus, which apply to messages, and contain zero or more types (corresponding to the arguments of the message).

5.3.1 Syntax

The language of type string is context free. It is also a prefix code, which is a property that is used by the recursive structure of the language itself.

[3] Compare with the *whence* parameter to the `lseek()` system call.

Type strings can be described by a non-ambiguous context free grammar (in which ε represents the empty string). With start symbol *type*:

type	⇒	*base_type* | *container_type*
base_type	⇒	**b** | **y** | **n** | **q** | **i** | **u** | **x** | **t** | **s** | **o** | **g**
container_type	⇒	**v** | **m** *type* | **a** *type* | **(** *types* **)** | **{** *base_type type* **}**
types	⇒	ε | *type types*

5.3.2 Semantics

The derivation used to obtain a type string from the given grammar creates an abstract syntax tree describing the type. The effect of deriving through each right hand side term containing a terminal is specified below:

b
This derivation corresponds to the boolean type.

y
This derivation corresponds to the byte type.

n
This derivation corresponds to the signed 16-bit integer type.

q
This derivation corresponds to the unsigned 16-bit integer type.

i
This derivation corresponds to the signed 32-bit integer type.

u
This derivation corresponds to the unsigned 32-bit integer type.

x
This derivation corresponds to the signed 64-bit integer type.

t
This derivation corresponds to the unsigned 64-bit integer type.

d
This derivation corresponds to the double precision floating point number type.

s
This derivation corresponds to the string type.

o
This derivation corresponds to the object path type.

g
This derivation corresponds to the signature type.

v
This derivation corresponds to the variant type.

m *type*
This derivation corresponds to the maybe type which has a value of `Nothing` or `Just` *x* for some *x* in the range of *type*.

a *type*
This derivation corresponds to the array type in which each element has the type *type*.

(*types*)
This derivation corresponds to the structure type that has the types expanded by *types*, in order, as its item types.

{ *base_type type* }
This derivation corresponds to the dictionary entry type that has *base_type* as its key type and *type* as its value type.

Chapter 6

Serialisation Format

This chapter describes the serialisation format that is used by GVariant. This serialisation format is newly developed and described for the first time here.

6.1 Related Work

Attempts were made to evaluate candidates for use as the serialisation format for GSettings (and therefore the serialisation format that GVariant would implement). Special consideration was paid to formats that are implemented by software that is already part of the GNOME desktop.

In the end, each format that was considered was found to conflict with the requirements given in Chapter 4.

For this reason, a new serialisation format was created. The documentation of this serialisation format is what forms the main body of this chapter.

6.1.1 DBus

Since GVariant is largely compatible with DBus, it would make sense to use the serialisation format of DBus (plus modifications where appropriate) as the serialisation format for GVariant.

To do so, however, would conflict with a number of requirements that were established for GVariant.

Most fundamentally, Performance Requirement 4 would be violated. DBus messages are encoded in such a way that in order to fetch the 100th item out of an array you first have to iterate over the first 99 items to discover where the 100th item lies. A side effect of this iteration would be a violation of Performance Requirement 3.

Additionally, using the DBus serialisation format with an API like that mandated by Context Requirement 4 would imply a violation of due to the fact that subparts of DBus messages can change meaning when subjected to different starting alignments. This is discussed in more detail in Section 6.3.3.

6.1.2 XML

As the current serialisation format of GConf, consideration was given to using XML (see [**XML**]) as the native serialisation format of GVariant.

Although XML does not implement the type system of DBus, per se, its flexibility as a file format provides the possibility of encoding DBus types and values.

As with the DBus serialisation format, however, the two main problems with using XML would be a violation of (particularly in the case that Context Requirement 3 is satisfied) and of Performance Requirement 4.

For these reasons, XML can not be used as the primary serialisation format of GVariant. GVariant, as implemented, does contain support for

storing values in an XML-like format for situations where performance is not important (see Section 8.6).

6.1.3 CORBA

GNOME currently makes significant (but decreasing) use of CORBA (see [**CORBA**]) as a framework for cross-process communication.

CORBA is targeted at RPC, and not to serialisation for purposes of persistent storage. It also brings its own incompatible type system (in violation of Context Requirement 6).

CORBA is very complicated and this complication is leading to a decrease in its usage among GNOME applications (which are rapidly switching to DBus for IPC). Even GConf (which was originally based on CORBA) has been ported to DBus.

For these reasons, CORBA is not seen as a viable option.

6.1.4 Protocol Buffers

Google has recently developed Protocol Buffers (see [**protobuf**]) as a solution to address some of the performance problems associated with XML. The performance is improved by a constant factor; there is no improvement in the asymptotic complexity of certain operations, as would be required to satisfy Performance Requirement 4.

When using Protocol Buffers, you specify the format of your data, ahead of time, in a `.proto` file. It takes this file and produces parser/printer code for your language of choice (among C++, Java and Python).

This approach is not suitable for use in a situation where a server process has to deal with data of arbitrary structure without knowing that structure in advance. This case is exactly the case of a configuration settings storage system.

6.2 Notation

Throughout this chapter a number of examples will be provided using a common notation for types and values.

The notation used for types is exactly the type strings described in Chapter 5.

The notation used for values will be familiar to users of either Python or Haskell. Arrays (lists) are represented with square brackets and structures (tuples) with parentheses. Commas separate elements. Strings are single-quoted. Numbers prefixed with `0x` are taken to be hexadecimal.

The constants `True` and `False` represent the boolean constants. The nullary data constructor of the maybe type is denoted `Nothing` and the unary one `Just`.

6.3 Concepts

GVariant value serialisation is a total and injective function from values to pairs of byte sequences and type strings. Serialisation is deterministic in that there is only one acceptable "normal form" that results from serialising a given value. Serialisation is non-surjective: non-normal forms exist.

The byte sequence produced by serialisation is useless without also having the type string. Put another way, deserialising a byte sequence requires knowing this type.

Naturally, we expect that deserialising the byte sequence resulting from serialising a value (using the same type string) will produce the the same value.

Before discussing the specifics of serialisation, there are some concepts that are pervasive in the design of the format that should be understood.

6.3.1 Byte Sequence

A byte sequence is defined as a sequence of bytes which has a known length. In all cases, in GVariant, knowing the length is essential to being able to successfully deserialise a value.

6.3.2 Byte Boundaries

Starting and ending offsets used in GVariant refer not to byte positions, but to byte boundaries. For the same reason that it is possible to have $n + 1$ prefixes of a string of length n, there are $n + 1$ byte boundaries in a byte sequence of size n.

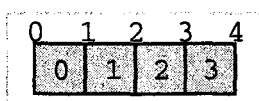

Figure 6.1: byte boundaries

When speaking of the start position of a byte sequence, the index of the starting boundary happens to correspond to the index of the first byte. When speaking of the end position, however, the index of the ending boundary will be the index of the last byte, plus 1. This paradigm is very commonly used and allows for specifying zero-length byte sequences.

6.3.3 Simple Containment

A number of container types exist with the ability to have child values. In all cases, the serialised byte sequence of each child value of the container will appear as a contiguous sub-sequence of the serialised byte sequence of that container — in exactly the same form as it would appear if it were on its own. The child byte sequences will appear in order of their position in the container.

It is the responsibility of the container to be able to determine the end (or equivalently, length) and start of each child element.

This property permits a container to be deconstructed into child values simply by referencing a subsequence of the byte sequence of the container as the value of the child which is an effective way of satisfying .

This property is not the case for the DBus serialisation format. In many cases (for example, arrays) the encoding of a child value of a DBus message will change depending on the context in which that value appears. As an example: in the case of an array of doubles, should the value immediately preceding the array end on an offset that is an even multiple of 8 then the array will contain 4 padding bytes that it would not contain in the event that the end offset of the preceding value were shifted 4 bytes in either direction.

6.3.4 Alignment

In order to satisfy requirement Context Requirement 3, we must provide programmers with a pointer that they can comfortably use. On many machines, programmers cannot directly dereference unaligned values, and even on machines where they can, there is often a performance hit.

For this reason, all types in the serialisation format have an alignment associated with them. For strings or single bytes, this alignment is simply 1, but for 32-bit integers (for example) the alignment is 4. The alignment is a property of a type — all instances of a type have the same alignment.

All aligned values must start in memory at an address that is an integer multiple of their alignment.

The alignment of a container type is equal to the largest alignment of any potential child of that container. This means that, even if an array of 32-bit integers is empty, it still must be aligned to the nearest multiple of 4 bytes. It also means that the variant type (described below) has an alignment of 8 (since it could potentially contain a value of any other type and the maximum alignment is 8).

6.3.5 Fixed Size

To avoid a lot of framing overhead, it is possible to take advantage of the fact that, for certain types, all instances will have the same size. In this case, the type is said to be a fixed-sized type, and all of its values are said to be fixed-sized values. Examples are a single integer and a tuple of an integer and a floating point number. Counterexamples are a string and an array of integers.

If a type has a fixed size then this fixed size must be an integer multiple of the alignment of the type. A type never has a fixed size of zero.

If a container type always holds a fixed number of fixed-size items (as in the case of some structures or dictionary entries) then this container type will also be fixed-sized.

6.3.6 Framing Offsets

If a container contains non-fixed-size child elements, it is the responsibility of the container to be able to determine their sizes. This is done using framing offsets.

A framing offset is an integer of some predetermined size. The size is always a power of 2. The size is determined from the overall size of the container byte sequence. It is chosen to be just large enough to reference each of the byte boundaries in the container.

As examples, a container of size 0 would have framing offsets of size 0 (since no bits are required to represent no choice). A container of sizes 1 through 255 would have framing offsets of size 1 (since 256 choices can be represented with a single byte). A container of sizes 256 through 65535 would have framing offsets of size 2. A container of size 65536 would have framing offsets of size 4.

There is no theoretical upper limit in how large a framing offset can be. This fact (along with the absence of other limitations in the serialisation format) allows for values of arbitrary size.

When serialising, the proper framing offset size must be determined by "trial and error" — checking each size to determine if it will work. It is possible, since the size of the offsets is included in the size of the container, that having larger offsets might bump the size of the container up into the next category, which would then require larger offsets. Such containers, however, would not be considered to be in "normal form". The smallest possible offset size must be used if the serialised data is to be in normal form.

Framing offsets are always stored at the end of containers and are unaligned. They are always stored in little-endian byte order.

Placing the unaligned framing offsets after the possibly-aligned data means that no bytes are ever wasted on padding. It also allows data to be written to a serialisation buffer "as you go" without first knowing the number of items that will be added to a container or the overall size of the container (two aspects which affect the amount of space required to store the offsets).

6.3.7 Endianness

Although the framing offsets of serialised data are always stored in little-endian byte order, the data visible to the user (via the interface mandated by requirement Context Requirement 3) is allowed to be in either big or little-endian byte order. This is referred to as the "encoding byte order". When transmitting messages, this byte order should be specified if not explicitly agreed upon.

The encoding byte order affects the representation of only 7 types of values: those of the 6 (16, 32 and 64-bit, signed and unsigned) integer types and those of the double precision floating point type. Conversion between different encoding byte orders is a simple operation that can usually be performed in-place (but see Section 7.1 for an exception).

6.4 Serialisation of Base Types

Base types are handled as follows:

6.4.1 Booleans

A boolean has a fixed size of 1 and an alignment of 1. It has a value of 1 for True or 0 for False.

6.4.2 Bytes

A byte has a fixed size of 1 and an alignment of 1. It may have any valid byte value. By convention, bytes are unsigned.

6.4.3 Integers

There are 16, 32 and 64-bit signed and unsigned integers. Each integer type is fixed-sized (to its natural size). Each integer type has alignment equal to its fixed size. Integers are stored in the encoding byte order. Signed integers are represented in two's complement.

6.4.4 Double Precision Floating Point

Double precision floating point numbers have an alignment and a fixed-size of 8. Doubles are stored in the encoding byte order.

6.4.5 Strings

Including object paths and signature strings, strings are not fixed-sized and have an alignment of 1. The size of any given serialised string is equal to the length of the string, plus 1, and the final serialised byte is a nul (0) terminator. The nul terminator is not strictly required (since the size is already known) but is provided as a convenience to C programs

that wish to access the string. The character set encoding of the string is not specified, but no nul byte is allowed to appear within the content of the string.

6.5 Serialisation of Container Types

Containers are handled as follows:

6.5.1 Variants

Variants are serialised by storing the serialised data of the child, plus a zero byte, plus the type string of the child. The reason the type string is stored at the end is the same reason framing offsets are stored at the end of container types — it ensures that no padding bytes are required.

The zero byte is required because, although type strings are a prefix code, they are not a suffix code. In the absence of this separator, consider the case of a variant serialised as two bytes — "ay". Is this a single byte, `'a'`, or an empty array of bytes?

6.5.2 Maybes

Maybes are encoded differently depending on whether their element type is fixed-sized not.

The alignment of a maybe type is always equal to the alignment of its element type.

6.5.2.1 Maybe of a Fixed-Sized Element

For the `Nothing` case, the serialised data is the empty byte sequence.

For the `Just` case, the serialised data is exactly equal to the serialised data of the child. This is always distinguishable from the `Nothing` case because all fixed-sized values have a non-zero size.

6.5.2.2 Maybe of a Non-Fixed-Sized Element

For the `Nothing` case, the serialised data is, again, the empty byte sequence.

For the `Just` case, the serialised form is the serialised data of the child element, followed by a single zero byte. This extra byte ensures that the `Just` case is distinguishable from the `Nothing` case even in the event that the child value has a size of zero.

6.5.3 Arrays

Arrays are said to be fixed width arrays or variable width arrays based on if their element type is a fixed-sized type or not. The encoding of these two cases is very different.

The alignment of an array type is always equal to the alignment of its element type.

6.5.3.1 Fixed Width Arrays

In this case, the serialised form of each array element is packed sequentially, with no extra padding or framing, to obtain the array. Since all fixed-sized values have a size that is a multiple of their alignment requirement, and since all elements in the array will have the same alignment requirements, all elements are automatically aligned.

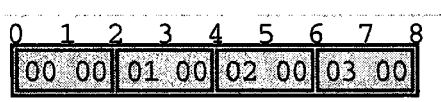

Figure 6.2: an array of 16-bit integers

The length of the array can be determined by taking the size of the array and dividing by the fixed element size. This will always work since all fixed-size values have a non-zero size.

6.5.3.2 Variable Width Arrays

In this case, the serialised form of each array element is again packed sequentially. Unlike the fixed-width case, though, padding bytes may need to be added between the elements for alignment purposes. These padding bytes must be zeros.

After all of the elements have been added, a framing offset is appended for each element, in order. The framing offset specifies the end boundary of that element.

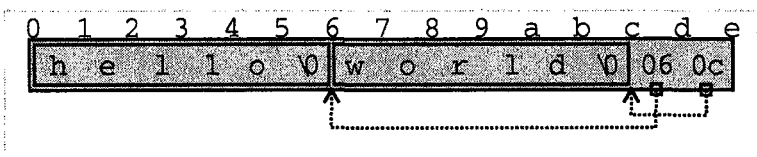

Figure 6.3: an array of strings

The size of each framing offset is a function of the serialised size of the array and the final framing offset, by identifying the end boundary of the final element in the array also identifies the start boundary of the framing offsets. Since there is one framing offset for each element in the array, we can easily determine the length of the array.

$$length = (size - last\_offset) / offset\_size$$

To find the start of any element, you simply take the end boundary of the previous element and round it up to the nearest integer multiple of the array (and therefore element) alignment. The start of the first element is the start of the array.

Since determining the length of the array relies on our ability to count the number of framing offsets and since the number of framing offsets is determined from how much space they take up, zero byte framing

offsets are not permitted in arrays, even in the case where all other serialised data has a size of zero. This special exception avoids having to divide zero by zero and wonder what the answer is.

6.5.4 Structures

As with arrays, structures are serialised by storing each child item, in sequence, properly aligned with padding bytes, which must be zero.

After all of the items have been added, a framing offset is appended, in reverse order, for each non-fixed-sized item that is not the last item in the structure. The framing offset specifies the end boundary of that element.

The framing offsets are stored in reverse order to allow iterator-based interfaces to begin iterating over the items in the structure without first measuring the number of items implied by the type string (an operation which requires time linear to the size of the string).

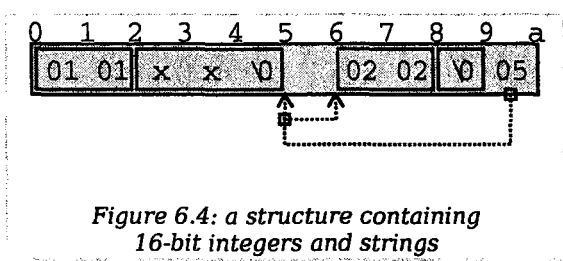

Figure 6.4: a structure containing 16-bit integers and strings

The reason that no framing offset is stored for the last item in the structure is because its end boundary can be determined by subtracting the size of the framing offsets from the size of the structure. The number of framing offsets present in any instance of a structure of a given type can be determined entirely from the type (following the rule given above).

The reason that no framing offset is stored for fixed-sized items is that their end boundaries can always be found by adding the fixed size to the start boundary.

To find the start boundary of any item in the structure, simply start from the end boundary of the nearest preceding non-fixed-size item (or from 0 in the case of no preceding non-fixed-sized items). From there, round up for alignment and add the fixed size for each intermediate item. Finally, round up to the alignment of the desired item.

For random access, it seems like this process can take a time linear to the number of elements in the structure, but it can actually be performed in a very small constant time. See Section 7.2.

If all of the items contained in a structure are fixed-size then the structure itself is fixed-size. Considerations have to be made to satisfy the constraints that are placed on the value of this fixed size.

First, the fixed size must be non-zero. This case would only occur for structures of the unit type or structures containing only such structures (recursively). This problem is solved by arbitrary declaring that the serialised encoding of an instance of the unit type is a single zero byte (size 1).

Second, the fixed sized must be a multiple of the alignment of the structure. This is accomplished by adding zero-filled padding bytes to the end of any fixed-width structure until this property becomes true. These bytes will never result in confusion with respect to locating framing offsets or the end of a variable-sized child because, by definition, neither of these things occur inside fixed-sized structures.

Figure 6.4 depicts a structure of type `(nsns)` and value `[257, 'xx', 514, '']`. One framing offset exists for the one non-fixed-sized item that is not the final item (namely, the string `'xx'`). The process of "rounding up" to find the start of the second integer is indicated.

6.5.5 Dictionary Entries

Dictionary entries are treated as structures with exactly two items — first the key, then the value. In the case that the key is fixed-sized, there will be no framing offsets, and in the case the key is non-fixed-size

there will be exactly one. As the value is treated as the last item in the structure, it will never have a framing offset.

6.6 Examples

This section contains some clarifying examples to demonstrate the serialisation format. All examples are in little endian byte order.

The example data is given 16 bytes per line, with two characters representing the value of each byte. For clarity, a number of different notations are used for byte values depending on purpose.

- `'A` shows that a byte has the ASCII value of `A` (65).
- `sp` shows that a byte is an ASCII space character (32).
- `\0` shows that a byte is a zero byte used to mark the end of a string.
- `--` shows that the byte is a zero-filled padding byte used as part of a structure or dictionary entry.
- `##` shows that the byte is a zero-filled padding byte used as part of an array.
- `@@` shows that the byte is the zero-filled padding byte at the end of a Just value.
- any two hexadecimal digits show that a byte has that value.

Each example specifies a type, a sequence of bytes, and what value this byte sequence represents when deserialised with the given type.

String Example
With type string `'s'`:

```
'h 'e 'l 'l  'o sp 'w 'o   'r 'l 'd \0
```

has a value of `'hello world'`.

Maybe String
With type string `'ms'`:

```
'h 'e 'l 'l  'o sp 'w 'o   'r 'l 'd \0 @@
```

has a value of `Just 'hello world'`.

Array of Booleans Example
With type string `'ab'`:

```
01 00 00 01  01
```

has a value of `[True, False, False, True, True]`.

Structure Example
With type string `'(si)'`:

```
'f 'o 'o \0  ff ff ff ff   04
```

has a value of `('foo', -1)`.

Structure Array Example
With type string `'a(si)'`:

```
'h 'i \0 --  fe ff ff ff   03 ## ## ##  'b 'y 'e \0
ff ff ff ff  04 09
```

has a value of `[('hi', -2), ('bye', -1)]`.

String Array Example
With type string `'as'`:

```
'i \0 'c 'a  'n \0 'h 'a   's \0 's 't  'r 'i 'n 'g
's '? \0 02  06 0a 13
```

has a value of `['i', 'can', 'has', 'strings?']`.

Nested Structure Example
With type string `'((ys)as)'`:

```
'i 'c 'a 'n  \0 'h 'a 's   \0 's 't 'r  'i 'n 'g 's
'? \0 04 05
```

has a value of `(('i', 'can'), ['has', 'strings?'])`.

Simple Structure Example
With type string `'(yy)'`:

```
70 80
```

has a value of `(0x70, 0x80)`.

Padded Structure Example 1
With type string `'(iy)'`:

```
60 00 00 00  70 -- -- --
```

has a value of `(96, 0x70)`.

Padded Structure Example 2
With type string `'(yi)'`:

```
70 -- -- --  60 00 00 00
```

has a value of `(0x70, 96)`.

Array of Structures Example
With type string `'a(iy)'`:

```
60 00 00 00  70 -- -- --   88 02 00 00  f7 -- -- --
```

has a value of `[(96, 0x70), (648, 0xf7)]`.

Array of Bytes Example
With type string `'ay'`:

```
04 05 06 07
```

has a value of `[0x04, 0x05, 0x06, 0x07]`.

Array of Integers Example
With type string `'ai'`:

```
04 00 00 00  02 01 00 00
```

has a value of `[4, 258]`.

Dictionary Entry Example
With type string `'{si}'`:

```
'a sp 'k 'e  'y \0 -- --   02 02 00 00 06
```

has a value of `{'a key', 514}`.

6.7 Non-Normal Serialised Data

Nominally, deserialisation is the inverse operation of serialisation. This would imply that deserialisation should be a bijective partial function.

If deserialisation is a partial function, something must be done about the cases where the serialised data is not in normal form. Normally this would result in an error being raised.

6.7.1 An Argument Against Errors

Requirement Performance Requirement 3 forbids us from scanning the entirety of the serialised byte sequence at load time; we can not check for normality and issue errors at this time. This leaves any errors that might occur to be raised as exceptions as the values are accessed.

Faced with the C language's poor (practically non-existent) support for exceptions and with the idea that any access to a simple data value might possibly fail, this solution also becomes rapidly untenable.

The only reasonable solution to deal with errors, given our constraints, is to define them out of existence. Accepting serialised data in non-normal form makes deserialisation a surjective (but non-injective) total function. All byte sequences deserialise to some valid value.

For security purposes, what is done with the non-normal values is precisely specified. One can easily imagine a situation where a content filter is acting on the contents of messages, regulating access to a security-sensitive component. If one could create a non-normal form of a message that is interpreted differently by the deserialiser in the filter and the deserialiser in the security-sensitive component, one could "sneak by" the filter.

6.7.2 Default Values

When errors are encountered during deserialisation, lacking the ability to raise an exception, we are forced into a situation where we must return a valid value of the expected type. For this reason, a "default value" is defined for each type. This value will often be the result of an error encountered during deserialisation.

One might argue that a reduction in robustness comes from ignoring errors and returning arbitrary values to the user. It should be pointed out, though, that for most types of serialised data, a random byte error is much more likely to cause the data to remain in normal form, but with a different value. We cannot capture these cases and these cases might result in any possible value of a given type being returned to the user. We are forced to resign ourselves to the fact that the best we can do, in the presence of corruption, is to ensure that the user receives some value of the correct type.

The default value for each type is:

Booleans
The default boolean value is False.

Bytes
The default byte value is nul.

Integers
The default value for any size of integer (signed or unsigned) is zero.

Floats
The default value for a double precision floating point number is positive zero.

Strings
The default value for a string is the empty string.

Object Paths
The default value for an object path is `'/'`.

Signatures
The default value for a signature is the nullary signature (ie: the empty string).

Arrays
The default value for an array of any type is the empty array of that type.

Maybes
The default value for a maybe of any type is the `Nothing` of that type.

Structures
The default value for a structure type is the structure instance that has for the values of each item, the default value for the type of that item.

Dictionary Entries
Similarly to structures, the default value for a dictionary entry type is the dictionary entry instance that has its key and value equal to their respective defaults.

Variants
The default variant value is the variant holding a child with the unit type.

It is worth noting that the default value for any fixed-sized type serialises to an all-zero byte sequence. This property simplifies the handling of these cases.

6.7.3 Handling Non-Normal Serialised Data

On a properly functioning system, non-normal values will not regularly be encountered, so once a problem has been detected, it is acceptable if performance is arbitrarily bad. For security reasons, however, untrusted data must always be checked for normality as it is being accessed. Due to the frequency of these checks, they must be fast.

Nearly all rules contained in this section for deserialisation of non-normal data keep this requirement in mind. Specifically, all rules can be decided in a small constant time (with a couple of very small exceptions). It would not be permissible, for example, to require that an array with an inconsistency anywhere among its framing offsets be treated as an empty array since this would require scanning over all of offsets (linear in the size of the array) just to determine the array size.

There are only a small number of different sorts of abnormalities that can occur in a serialised byte sequence. Each of them, along with what to do, is addressed in this section.

The following list is meant to be a definitive list. If a serialised byte sequence has none of these problems then it is in normal form. If a serialised byte sequence has any of these problems then it is not in normal form. Examples will be given in Section 6.7.4.

Wrong Size for Fixed Sized Value
In the event that the user attempts deserialisation using the type of a fixed-width type and a byte sequence of the wrong length, the default value for that type will be used.

Non-zero Padding Bytes

This abnormality occurs when any padding bytes are non-zero. This applies for arrays, maybes, structures and dictionary entries. This abnormality is never checked for — child values are deserialised from their containers as if the padding was zero-filled.

Boolean Out of Range

In the event that a boolean contains a number other than zero or one it is treated as if it were true. This is for purpose of consistency with the user accessing an array of booleans directly in C. If, for example, one of the bytes in the array contained the number 5, this would evaluate to True in C.

Possibly Unterminated String

If the final byte of the serialised form of a string is not the zero byte then the value of the string is taken to be the empty string.

String with Embedded Nul

If a string has a nul character as its final byte, but also contains another nul character before this final terminator, the value of the string is taken to be the part of the string that precedes the embedded nul. This means that obtaining a C pointer to a string is still a constant time operation.

Invalid Object Path

If the serialised form of an object path is not a valid object path followed by a zero byte then the default value is used.

Invalid Signature

If the serialised form of a signature string is not a valid DBus signature followed by a zero byte then the default value is used.

Wrong Size for Fixed Sized Maybe

In the event that the size of a maybe instance with a fixed element size is not exactly equal to the size of that element, then the value is taken to be `Nothing`.

Wrong Size for Fixed Width Array
In the event that the serialised size of a fixed-width array is not an integer multiple of the fixed element size, the value is taken to be the empty array.

Start or End Boundary of a Child Falls Outside the Container
If the framing offsets (or calculations based on them) indicate that any part of the byte sequence of a child value would fall outside of the byte sequence of the parent then the child is given the default value for its type.

End Boundary Precedes Start Boundary
If the framing offsets (or calculations based on them) indicate that the end boundary of the byte sequence of a child value precedes its start boundary then the child is given the default value for its type.

The end boundary of a child preceding the start boundary may cause the byte sequences of two or more children to overlap. This error is ignored for the other children. These children are given values that correspond to the normal deserialisation process performed on these byte sequences with the type of the child.

If children in a container are out of sequence then it is the case that this abnormality is present. No other specific check is performed for children out of sequence.

Child Values Overlapping Framing Offsets
If the byte sequence of a child value overlaps the framing offsets of the container it resides within then this error is ignored. The child is given a value that corresponds to the normal deserialisation process performed on this byte sequence (including the bytes from the framing offsets) with the type of the child.

Non-Sense Length for Non-Fixed Width Array
In the event that the final framing offset of a non-fixed-width array points to a boundary outside of the byte sequence of the array, or indicates a non-integral number of framing offsets is present in the array, the value is taken to be the empty array.

Insufficient Space for Structure Framing Offsets
In the event that a serialised structure contains an insufficient space to store the requisite number of framing offsets, the error is silently ignored as long as the item that is being accessed has its required framing offsets in place. An attempt to access an item that requires an offset beyond those available will result in the default value.

6.7.4 Examples

This section contains some clarifying examples to demonstrate the proper deserialisation of non-normal data.

The byte sequences are presented in the same form as for the normal-form examples. A brief description is provided for why a value deserialises to the given value.

Wrong Size for Fixed Size Value
With type string `'i'`:

```
07 33 90
```

has a value of `0`.

Since any value with a type of `'i'` should have a serialised size of 4, and since only 3 bytes are given, the default value of zero is used instead.

Non-zero Padding Bytes
With type string `'(yi)'`:

```
55 66 77 88  02 01 00 00
```

has a value of `(0x55, 258)`.

Non-zero padding bytes (66 77 88) are simply ignored.

Boolean Out of Range

With type string `'ab'`:

```
01 00 03 04  00 01 ff 80   00
```

has a value of `[True, False, True, True, False, True, True, True, False]`.

Any non-zero booleans are treated as `True`.

Unterminated String

With type string `'as'`:

```
'h 'e 'l 'l  'o sp 'w 'o   'r 'l 'd \0  0b 0c
```

has a value of `['', '']` (two empty strings).

The second string deserialises normally as a single nul character, but the first string does not contain a nul character. Regardless of the fact that a nul character immediately follows it, the first string is replaced with the empty string (the default value for strings).

String with Embedded Nul

With type string `'s'`:

```
'f 'o 'o \0  'b 'a 'r \0
```

has a value of `'foo'`.

String with embedded nul but none at end

With type string `'s'`:

```
'f 'o 'o \0  'b 'a 'r
```

has a value of `''` (the empty string).

The last byte in the string is always checked to determine if there is a nul and, if not, the empty string is used as the value. This includes the case where a nul is present elsewhere in the string.

Wrong size for fixed-size maybe

With type string `'mi'`:

```
33 44 55 66  77 88
```

has a value of `Nothing`.

The only possible way for a value with type `'mi'` to be Just is for its serialised form to be exactly 4 bytes.

Wrong size for fixed-width array

With type string `'a(yy)'`:

```
03 04 05 06 07
```

has a value of `[]`.

With each array element as a pair of bytes, the serialised size of the array should be a multiple of two. Since this is not the case, the value of the array is the empty array.

Start or end boundary of child falls outside the container

With type string `'(as)'`:

```
'f 'o 'o \0  'b 'a 'r \0   'b 'a 'z \0  04 10 0c
```

has a value of `['foo', '', '']`.

No problems are encountered while unpacking the first element in the array (which is marked as falling between byte boundaries 0 and 4). When unpacking the 2nd element, its end offset (16) is outside of the bounds of the array. This offset (16) is also the start of the 3rd array element. As a result, both of these elements are given their default value (the empty string).

End boundary precedes start boundary

With type string `'(as)'`:

```
'f 'o 'o \0  'b 'a 'r \0   'b 'a 'z \0  04 00 0c
```

has a value of `['foo', '', 'foo']`.

Again, no problems are encountered while unpacking the first element in the array. When unpacking the second element it is noticed that the end boundary precedes the start. Since this is impossible, the default value of `''` is used instead. Unpacking the final element (from 0 to 12) occurs without problem. The final element overlaps the first element, however, and when assessing its value, the embedded nul character causes it to be cut off at `'foo'`.

Insufficient space for structure framing offsets

With type string '(ayayayayay)':

```
03 02 01
```

has a value of ([3], [2], [1], [], []).

Since this is not a fixed-size value, the fact that it has an impossible size does not cause it to receive its default value (ie: there is no concept of "minimum-size"). Unpacking the first three items in the structure occurs without a problem (demonstrating that the content of a value can overlap the framing offsets). Attempting to unpack the last two items fails, however, since the required framing offsets simply do not exist. The default values are used instead.

Chapter 7

Implementing the Format

This chapter contains information about the serialisation format that is not part of its specification.

This information discusses issues that will arise during implementation of the serialisation format. Certainly, the issues discussed in this chapter have had an impact on the GVariant implementation discussed in Chapter 10.

An unfortunate observation is made about the safety of byteswapping operations and a method is given (along with proof of correctness) that random accesses to the contents of a structure can be made in constant time, despite the fact that framing offset are omitted for fixed-sized values.

7.1 Notes on Byteswapping

Implementors may wish to perform in-place byteswapping of serialised GVariant data. There are a couple of things to consider in this case.

The primary concern arises from the fact that if non-normal serialised data is present then byteswapping may not be possible.

With a type string of `(ssn)` consider the following non-normal serialised data in little-endian byte order:

```
78 00 00 02
```

The first string has a length of 2 (including the nul terminator) and a value of `'x'`. The second string is given its default value of `''` as a result of its end offset of 0 preceding its start offset of 2. Finally, the 16-bit integer, with a start offset of 0 (thus overlapping the first string) has a value of `0x78`. The value of the entire structure is `('x', '', 120)`.

To change this serialised data to be in big-endian byte order requires the swapping of the bytes of the 16-bit value. To do so, however, would also modify the value of the string which these bytes overlap. In this case (and in general) there is no way to avoid this problem.

Because of this problem, any implementation wishing to perform in-place byteswapping of serialised data must first ensure that the data is in normal form.

There are a couple of cases where this requirement for normal form does not exist. In the case of any fixed-sized value or variable sized array, no framing offsets are present. This effectively eliminates the possibility of overlapping data and means that this cases can be byteswapped in-place without first checking for normality.

Through a fortunate alignment of circumstances, these types (together with strings, which need not be byteswapped at all) are exactly the sorts of data that an implementation may wish to make available to the user via a pointer. As a result it is easy to imagine that an implementation may end up not requiring the ability to in-place byteswap serialised data except in cases where it is always safe.

7.2 Calculating Structure Item Addresses

In the C language, structures exist in much the same way as they exist in the serialisation format. Each item in the structure follows the one preceding it as closely as possible, subject to alignment constraints.

No matter what is done, it is impossible to determine the address of an item in a structure in C in a constant amount of time. The sizes and alignments of the items preceding it each need to be considered — a process which can not occur in less than linear time. The algorithm for doing this is to start at the starting address of the structure and then for each preceding item in the structure, round up to its alignment requirement and add its size. Finally, round up to the alignment requirement of the item to be accessed.

This process can be described with a simple algebra containing two types of operations:

- $(+c)$: add to a natural number, some constant, c.
- $(\uparrow c)$: "align" (round up) a natural number up to the nearest multiple of some constant power of two, 2^c.

Assume that the compiler aligns integer values to their size. To find the address of a 32-bit integer following a 16-bit integer following an array of three 64-bit integers, for example, the following computation must be performed, given the address of the start of the structure, s:

$$((\uparrow 3); (+24); (\uparrow 1); (+2); (\uparrow 2))\ s$$

in which $(a; b)$ denotes reverse function composition: "a then b".

Of course, no sane C compiler saves this computation to be performed at each access. Instead, the compiler performs the computation at the time of the structure definition and builds a table containing the starting offset and size of each item in the structure. Because every item in the structure is of a fixed size and because the start address of the structure is always appropriately aligned, the address of an item in a structure

can always be specified as a constant relative to the address of the start of that structure.

For our example:

$$(+28)\ s$$

Admitting non-fixed-sized items to structures very obviously prevents the starting offset of items following any non-fixed-sized item from being a constant relative to the start of the structure. The start address of any item will clearly depend on the end address of the non-fixed-sized item that most immediately precedes it. Worse than this though, due to the fact that this end address has no particular alignment, the starting offset of each item cannot be expressed as a constant offset, even to the end of the non-fixed-sized item preceding it.

Without discovering another method to build a table, the address computation would have to be performed, in full, at each access - in linear time. Fortunately, another method exists, permitting constant-time access to structure members. It is possible to build a table with each row containing four integers such that this table permits calculating the start address of any structure item to be performed in only four operations:

$$((+a);\ (\uparrow b);\ (+c))\ \mathit{offsets}[i]$$

Where *offsets* is the array of framing offsets for the structure and i, a, b and c are the four integers from the table. By definition, $\mathit{offsets}[-1] = 0$.

7.2.1 Performing the Reduction

In this and the following sections, $(x \uparrow y)$ is the result of applying $(\uparrow y)$ to x. If x and y are constants then $(x \uparrow y)$ will also be a constant — allowing us to compute its value ahead of time.

Essentially, we are interested in a process by which we can reduce any length of sequence of constant adding and alignment operations to a sequence of length 3, with the form shown above. We can then perform

this small constant number of operations at each access instead of the full computation.

This reduction process occurs according to the following reduction rules (which are proven in Section 7.2.5):

Addition rule
$(+a); (+b) \Rightarrow (+(a + b))$

Greater alignment rule
$(\uparrow a); (+b); (\uparrow c) \Rightarrow (+(b \uparrow a)); (\uparrow c)$, if $c \geq a$

Lesser alignment rule
$(\uparrow a); (+b); (\uparrow c) \Rightarrow (\uparrow a); (+(b \uparrow c))$, if $c \leq a$

We can prove that, using these rules, any sequence of operations can be reduced to have no more than one alignment operation. If there exist two alignment operations in the sequence, one of these cases must be true:

- two alignment operations separated by exactly one addition
- two adjacent alignment operations
- two alignment operations separated by more than one addition

In the case that there is exactly one addition separating our two alignment operations then the greater or the lesser alignment rule may be immediately applied to reduce the number of alignment operations by one.

In the case that there are more than one additions, they can be merged down to a single addition by application of the addition rule before applying one of the alignment rules. In the case of two adjacent alignment operations, a (+0) operation can be introduced between then before applying one of the alignment rules.

Since we can reduce any sequence of operations to a sequence containing only one alignment operation, we can further reduce it to

the form $(+a); (\uparrow b); (+c)$ by using the addition rule to merge all of the additions that occur before and after this single alignment operation.

7.2.2 Computing the Table

Based on the reduction rules above, an efficient (but still linear time) algorithm for computing the entire table at once can be developed.

At all times, the "state so far" is kept as the four variables: i, a, b and c such that getting to the current location is possible by computing $((+a); (\uparrow b); (+c))$ relative to the $offset[i]$. i is kept equal to the index of the framing offset which specifies the end of the most recently encountered non-fixed-sized item in the structure (or -1 in the case that no such item has been encountered). a, b, c start at 0.

Three merge rules are defined to allow any additional operation to be appended to this sequence without changing the size of the form of the sequence; the merge rules effect only the integer values of a, b and c.

1. appending an alignment d less than or equal to the current alignment: $(a, b, c) := (a, b, c \uparrow d)$ as a direct result of the lesser alignment rule application $(+a); (\uparrow b); (+c); (\uparrow d) \Rightarrow (+a); (\uparrow b) (+c \uparrow d)$.

2. appending an alignment d greater than the current alignment: $(a, b, c) := (a + (c \uparrow b), d, 0)$ by the greater alignment rule application $(+a); (\uparrow b); (+c); (\uparrow d) \Rightarrow (+a); (+c \uparrow b); (\uparrow d)$, addition rule application to $(+a + (c \uparrow b)); (\uparrow d)$ and harmless appending of $(+0)$ to give $(+a + (c \uparrow b)); (\uparrow d); (+0)$.

3. appending an addition e: $(a, b, c) := (a, b, c + e)$ by obvious use of the addition rule $(+a); (\uparrow b); (+c); (+e) \Rightarrow (+a); (\uparrow b); (+(c + e))$.

Each time a non-fixed-sized item is encountered, i is incremented and a, b, c are set back to zero.

The algorithm is implemented by the following Python function which takes a list of (alignment, fixed size) pairs as input, representing the structure items. Its output is the table, given as an array of 4-tuples.

```
def generate_table (items):
  (i, a, b, c) = (-1, 0, 0, 0)
  table = []
  for (d, e) in items:
    if d <= b:
      (a, b, c) = (a, b, align(c, d))     # merge rule #1
    else:
      (a, b, c) = (a + align(c, b), d, 0) # merge rule #2
    table.append ((i, a, b, c))
    if e == -1:                           # item is not fixed-sized
      (i, a, b, c) = (i + 1, 0, 0, 0)
    else:
      (a, b, c) = (a, b, c + e)           # merge rule #3
  return table
```

It is assumed that `align(a, b)` computes $(a \uparrow b)$.

7.2.3 Further Reduction

The reductions described above are non-confluent. An equivalence on the final sequence of operations exists. Specifically, if d is a multiple of 2^b, then:

$$(+a); (\uparrow b); (+(c + d)) = (+(a + d)); (\uparrow b); (+c)$$

This is because, being a multiple of 2^b, d can "pass through" the alignment operation without change.

Consider, for example, the following:

$$(n + 16) \uparrow 3$$

It is clear that this is equivalent to

$$(n \uparrow 3) + 16$$

since there are no low order bits in the binary representation of 16 to be affected by a rounding operation that clears only the bottom 3 bits.

In the case where only small alignment constraints are encountered (no larger than 8) it is possible (by shifting multiples of 256 out of c into a) to ensure that c fits into no more than a single byte. This applies to the serialisation format as specified, considering that the largest alignment constraint ever encountered is 3.

7.2.4 Plus/And/Or Representation

As a micro-optimisation, after performing the reduction in the previous section, the resulting values of a, b, c can be transformed such that the calculation can be performed in only 3 commonly-available machine instructions.

This transformation takes advantage of three simple facts about rounding.

First note that rounding up to the nearest multiple of any number is the same as adding that number, minus 1, then rounding down to the nearest multiple of that number.

Second, note that rounding down to the nearest multiple of a number that is a power of two is the same as taking the bitwise and with the bitwise complement of that number minus 1.

Third, note that the result of rounding to a multiple of a power of 2 results in the low order bits of the result being cleared. Adding a number less than that multiple to the result of the rounding can't possibly result in carrying, so using bitwise or is an equivalent operation.

Keeping in mind that after the reduction in the last section, $c < 2^b$:

$$((+a); (\uparrow b); (+c)\ s) = ((+ (a + 2^b - 1)); (\& \sim(2^b - 1)); (|c))\ s)$$

where | denotes bitwise or, & denotes bitwise and, and ~ denotes bitwise complement.

We can therefore choose to store the following into the table:

$$(a + 2^b - 1, \sim(2^b - 1), c)$$

and for each address we calculate, we are only required to perform an addition, a bitwise and and a bitwise or.

7.2.5 Proof of Reduction Rules

Given a few "intuitive" lemmas, we can prove that the reduction rules are sound.

Lemma 1
$\forall a, b: (\uparrow a); (\uparrow b) = (\uparrow (max(a, b)))$

since alignment is always to powers of two, two successive alignment operations are equivalent to the "most powerful" of the two.

Lemma 2
$\forall a, b, c, r: r = (\uparrow c) \Rightarrow r(a) + r(b) = r(a + r(b))$

since r(b) is already a multiple of 2c it can "pass through" the second application of r without change.

Lemma 3
$\forall c, (0 \uparrow c) = 0$

7.2.5.1 Addition Rule

Associativity of addition:

$$\forall a, b, n: (n + a) + b = n + (a + b)$$

which is just the same as:

$$\forall a, b, n: ((+a); (+b))\ n = (+(a + b))\ n$$

By partial instantiation:

$$\forall n: ((+a); (+b))\ n = (+(a + b))\ n$$

and then by extensionality:

$$(+a); (+b) = (+(a + b))$$

7.2.5.2 Greater Alignment Rule

Let $r = (\uparrow c)$ and $s = (\uparrow a)$.

Lemma 2:

$$\forall m, n : s(n) + s(m) = s(s(n) + m)$$

Lemma 3 allows:

$$\forall m, n : s(n) + s(m) + s(0) = s(s(n) + m)$$

Repeated application of lemma 2 to the above:

$$\forall m, n : s(n) + s(s(m) + 0) = s(s(n) + m)$$

$$\forall m, n : s(s(n) + s(m) + 0) = s(s(n) + m)$$

Which of course is equivalent to:

$$\forall m, n : s(s(n) + s(m)) = s(s(n) + m)$$

Since addition commutes and we universally quantify over both m and n, there is no reason that what works for one won't work equally well for the other:

$$\forall m, n: s(s(n) + s(m)) = s(n + s(m))$$

so, clearly:

$$\forall m, n: s(s(n) + m) = s(n + s(m))$$

Which we can partially instantiate as:

$$\forall n: s(s(n) + b) = s(n + s(b))$$

It must be true, then, that:

$$\forall n: r(s(s(n) + b)) = r(s(n + s(b)))$$

Remembering that $r = (\uparrow c)$ and $s = (\uparrow a)$:

$$\forall n: ((\uparrow a); (\uparrow c))\ ((n \uparrow a) + b) = ((\uparrow a); (\uparrow c))\ (n + (b \uparrow a))$$

And lemma 1 (since $a \leq c$) merges this into:

$$\forall n: (\uparrow c)\ ((n \uparrow a) + b) = (\uparrow c)\ (n + (b \uparrow a))$$

$$\forall n: ((\uparrow a); (+b); (\uparrow c))\ n = ((+(b \uparrow a)); (\uparrow c))\ n$$

By extensionality:

$$(\uparrow a); (+b); (\uparrow c) = (+(b \uparrow a)); (\uparrow c)$$

7.2.5.3 Lesser Alignment Rule

Let $r = (\uparrow a)$ and $s = (\uparrow c)$.

Trivially:

$$\forall n: s(r(n) + b) = s(r(n) + b)$$

From lemma 1, since $c \leq a$:

$$\forall n: s(s(r(n)) + b) = s(r(n) + b)$$

Then lemma 2 allows:

$$\forall n: s(r(n)) + s(b) = s(r(n) + b)$$

Effectively reversing the first application of lemma 1:

$$\forall n\colon r(n) + s(b) = s(r(n) + b)$$

Remembering $r = (\uparrow a)$ and $s = (\uparrow c)$:

$$\forall n\colon ((+(b \uparrow c)); (\uparrow a))\ n = ((\uparrow a); (+b); (\uparrow c))\ n$$

By extensionality:

$$(+(b \uparrow c)); (\uparrow a) = (\uparrow a); (+b); (\uparrow c)$$

PART III

This part documents the implementation of GVariant that was developed for inclusion in the GNOME platform. The application programmer interface is introduced and some examples are given of common use cases to illustrate the basic functioning of this implementation. Some noteworthy internal implementation details are described.

Chapter 8

Programmer Interface

As a first step to understanding the implementation of GVariant, this chapter gives an overview of its interfaces.

The chapter is split into a number of sections such that each major "part" of the interface is briefly described in its own section.

Detailed API reference documentation, on a call-by-call basis is provided as Appendix A.

8.1 Types

As GVariant has been developed in, and for, the C programming language, the possibility of performing any sort of compiler supported static type checking of programs using GVariant is practically non-existent.

For this and other reasons, GVariant must feature some notion of representing types, on its interfaces, as runtime objects. The name of the type of this runtime object is `GVariantType`.

The range of `GVariantType` includes all of the types described in Chapter 5. `GVariantType` also includes "wildcard types".

8.1.1 Wildcard Types

`GVariantType` supports the concept of "matching" and "wildcard types". Any type matches itself, but a wildcard type matches other types as well. A wildcard type can never be the type of an instance (but an instance, of course, may match a wildcard type).

Matching can be used to support a degree of polymorphism while enforcing runtime type assertions.

There are three base wildcard types which each match a number of other types. An infinite number of wildcard types can be formed by applying the other type constructors, in the usual way, to other wildcard types.

All wildcard
The all wildcard type matches any type. This wildcard is represented by the `'*'` character.

Basic wildcard
The basic wildcard type matches any one of the basic types. It is itself considered to be a basic type. This wildcard is represented by the `'?'` character.

Structure wildcard
The structure wildcard type matches any structure type. This wildcard is represented by the `'r'` character.

The grammar of type strings is expanded to support wildcard types in the following manner. At any position in the derivation of a type string where a valid type string could appear then any of the new terminal characters (`'*'`, `'?'` or `'r'`) can now appear. At any position of a type string where only a base type could appear (ie: as the key of a dictionary entry type) the terminal character `'?'` can now appear.

The reason that a base wildcard type exists to match any structure type, but not for any of the other type constructors, is because structures are the only variadic type constructor. Equivalent wildcard types may

be formed for the other type constructors by providing the other base wildcard types as arguments for those constructors (consider type string `'a*'` and `'{?*}'`, for example).

8.1.2 Type Classes

A user of GVariant may be interested in handling GVariant instances of arbitrary types. This can be done by recursing and iterating over the structure of the instance in a generic way.

In order for the user to be able to unwind the structure of the type of a value, they need the ability to categorise many different types into classes. One step of a recursive algorithm can then perform its local task based on which class it is dealing with.

To this end, a small finite number of "type classes" exist as an enumerated a type called `GVariantTypeClass`.

Querying the type class of a `GVariantType` effectively gives all of the information about the "top layer" of the type. In terms of type strings, the type class can always be determined by looking only at the first character.

Some examples of type classes are "array", "structure" and "32 bit signed integer".

8.2 Values

The most central part of the interface, of course, is the type that represents a single value. This type is, accordingly, given the name `GVariant`.

A `GVariant` is essentially a dependent pair of one of the types described in Chapter 5 and a value of that type. It is no accident that this definition bears close resemblance to the definition given for the "variant" type in the same chapter — these two things serve the same purpose and can

contain exactly the same range of values. The only difference is that one exists within the type system of GVariant and the other exists within C.

A `GVariant` has its type and value when it is constructed and these two things never change. A `GVariant` is a value, not a variable. The only way to "change" the value is to destroy the instance create a new one in its place.

Each `GVariant` is reference counted. This allows a number of users to "share" a value. So long as a particular user holds on to their reference they can ensure that the value will continue to exist. Coupled with the statement in the previous paragraph, this enables a given user to be certain that the value that they have a reference to will never change.

Floating reference counts are supported. This concept is very familiar to GNOME programmers, as the GObject type system features it. In essence, in addition to a reference count, each instance has a "floating" flag. A new "sink" operation is defined as follows: if the floating flag is set, then unset it and do nothing else; if the floating flag is unset then increase the reference count.

When any `GVariant` instance is added to a container, the sink operation is performed on that instance. In the case that the floating flag was not set, this causes the container to acquire a new reference to the instance. New instances are created with the floating flag set, however. This allows for the programmer to skip the step of explicitly releasing their reference to an instance in the very common case that it is created only to be added directly to a container (since at the point of adding to the container the sink operation effectively transfers the reference from the caller to the container). This feature is a nice convenience in a language that lacks automatic reference counting.

8.3 Plain C Interfaces

Creating new instances or gaining access to the value of an existing instance is accomplished by a range of calls that are detailed in the appendix.

For each base type `x` there is a `g_variant_new_x()` function and a `g_variant_get_x()` function.

For each container type class there is a function to allow construction of a `GVariant` instance with a type of that class, given existing child instances.

There are also `g_variant_n_children()` and `g_variant_get_child()` calls which may be applied to children in obvious ways to loop over their contents.

As an added convenience there are iterator and builder interfaces that allow for step-by-step construction and deconstruction of containers.

Finally, there are calls to allow direct pointer access to serialised data that can be directly understood by C. The functions `g_variant_get_fixed()` and `g_variant_get_fixed_array()` can be applied to fixed-sized values and arrays thereof.

8.4 varargs C Interfaces

Even with the convenience functions provided as part of the builder and iterator interfaces, constructing and deconstructing complex hierarchies of values (particularly complex structure types) can be particularly frustrating. For this reason, a `printf()`-style interface has been introduced.

The functions `g_variant_new()` and `g_variant_get()` each accept a special format string. This format string describes the types of arguments that will be collected and the final (or initial) type of the value being constructed (or deconstructed).

Any type string (including those containing wildcards) is a valid format string. Additionally any type string appearing within the format string may have `'@'` prepended to it. Any type string corresponding to a fixed-type that appears within the format string may have `'&'` prepended to it.

These two modifiers don't change the type involved in the construction or deconstruction but change how the arguments are collected.

More detail about format strings is provided in the appendix.

As with all varargs functions, this interface is slightly evil. This evilness, however, provides for significantly less typing.

Versions of these functions that take a pointer to a `va_list` (similar to `vprintf()`) also exist. They are denoted with the suffix `_va`.

8.5 Load and Store

Contained in a separate header file and not intended for use by "normal users" is support for interfacing with GVariant on the level of serialised data.

Calls exist for writing serialised data out to a buffer, or for requesting the serialised data stored internally within a GVariant (in the case that this does not yet exist, it will be created).

Calls also exist for creating new GVariant instances from serialised data — either by taking a copy of the data, or in the most efficient case using the data in-place.

8.6 Markup

Facilities are provided for pretty-printing `GVariant` instances to and parsing them from a GMarkup[1]-based format.

[1] GMarkup is a substantial subset of XML designed for simplicity. It has all of the basic features which one would normally recognise as being XML. Some missing features include user-defined entities, DTD validation and character encodings other than UTF-8.

Any value is representable in the markup language, so parsing composed with printing is an identity operation. One current exception (which may be addressed in future work) is a loss of floating point precision.

Support has been added[2] to GLib to support "subparsing" of GMarkup documents whereby a subparser can be invoked to handle a segment of a larger document. It is in this context, as a subparser, that GVariant markup parsing is expected to be most commonly used.

An example document in this markup language is given below. The type string of the type of the value that results from parsing the document is `(yasabaq)`.

```
<struct>
  <byte>42</byte>

  <array>
    <string>hello</string>
    <string>world</string>
  </array>

  <array>
    <true/>
    <true/>
    <false/>
  </array>

  <!-- GVariant is unable to infer the type
       so it must be explicitly specified
    -->
  <array type='aq'/>
</struct>
```

[2] http://bugzilla.gnome.org/show_bug.cgi?id=337518#c24

Chapter 9

Clarifying Examples

Before describing the implementation of GVariant in more detail, this chapter provides some insight into what occurs, internally, in response to some common usage scenarios.

9.1 Reading from a mapped file

The first example demonstrates how GVariant can be used to access a single string within a memory mapped file containing an array of strings.

First, we assume that a `GMappedFile` named *mapped* exists.

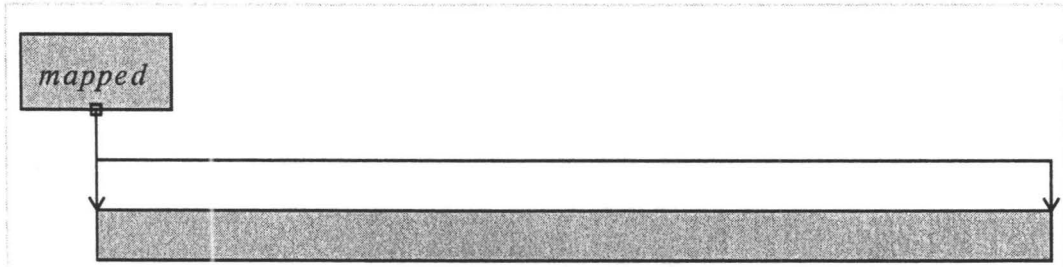

Figure 9.1: a `GMappedFile`

The mapped file contains, among its data, the serialised form of an array of strings. We can tell GVariant to create an instance representing the value of this array by calling `g_variant_from_data()`.

```
GVariant *array;

array = g_variant_from_data (G_VARIANT_TYPE ("as"),
                             mapped_data + offset,
                             size,
                             G_VARIANT_TRUSTED,
                             g_mapped_file_free,
                             mapped);
```

This function call causes a number of things to happen.

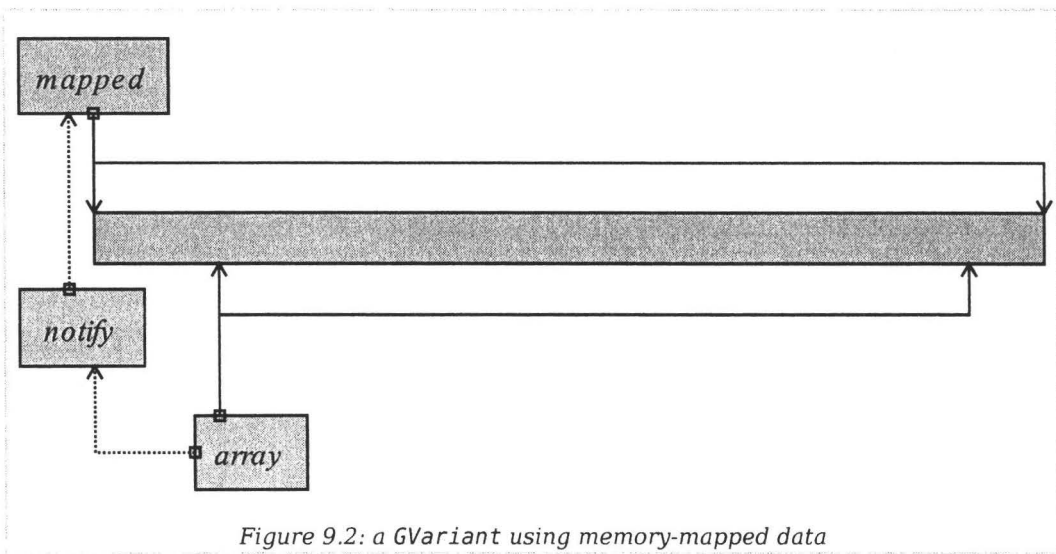

Figure 9.2: a `GVariant` using memory-mapped data

Most obviously, a new instance will be created to represent the array. This instance contains a pointer to the serialised data of the array (including its size). At this point, there has been no access to the contents of the mapped file — merely an exchange of pointers.

Of course, since GVariant didn't take its own copy of the data, GMappedFile must continue to exist for the duration of the life

of *array*. This is the purpose of the last two arguments to `g_variant_from_data()`. This is where the *notify* instance comes in.

notify is internal to GVariant and is never made accessible to the programmer. Its purpose is to exist for as long as the data provided by the user is needed. When it stops existing, it calls a notification function. In this case, the notification function is the freeing of the mapped file and its data.

There are two reasons that this indirect approach has been used instead of embedding the notify closure directly into the array itself. The first is that there simply wasn't enough room to store two extra pointers into a `GVariant` instance. The second is that having the notify instance provides for more flexibility.

Imagine now that we want to actually obtain one of the strings from inside the string array, the one with an index of 2.

```
GVariant *string;

string = g_variant_get_child (array, 2);
```

Strings are variable-sized, so in order to determine where our string lies among the serialised data we have to read the framing offsets associated with it. We read two (consecutive) integers telling us the start and the end of the string's data and use these offsets as the pointers for a new instance. No other data is read at this point.

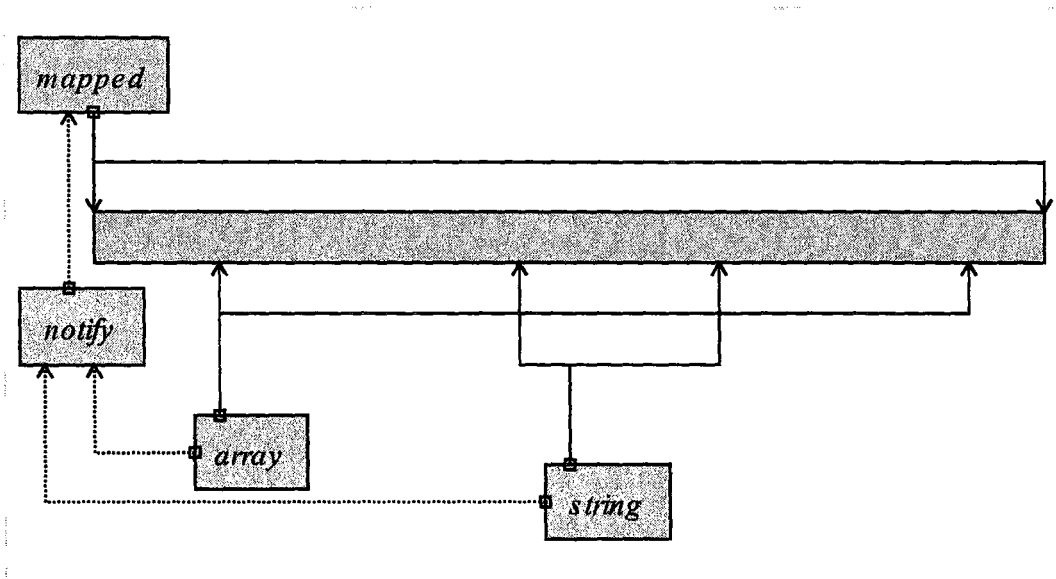

Figure 9.3: a GVariant shares the memory of its parent

Notice that *string* directly references the notify instance now. This means that if we were to drop our reference to the array it could be freed.

```
g_variant_unref (array);
```

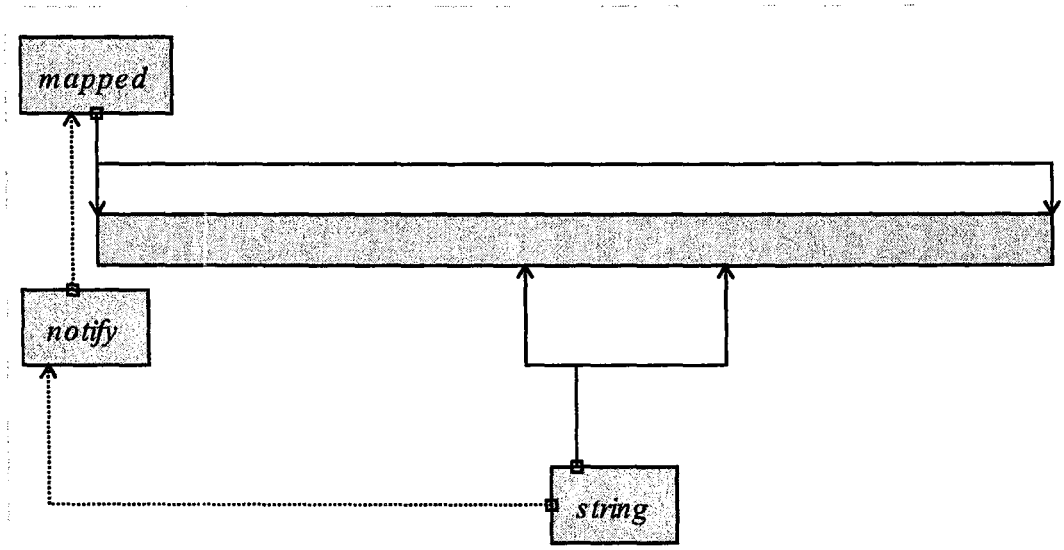

Figure 9.4: source data kept alive as long as it is in use

At this point, we might want to actually access the string data. This is a very simple proposition.

```
const gchar *ptr;
gsize len;

ptr = g_variant_get_string (string, &len);
```

Because we provided the `G_VARIANT_TRUSTED` flag when loading the data we know that the string is properly formatted and of the right length. This allows GVariant to provide the length of the string to the programmer for free. We're also confident that the string is properly nul-terminated, so we don't have to access the string's data at all at this point — only return a pointer.

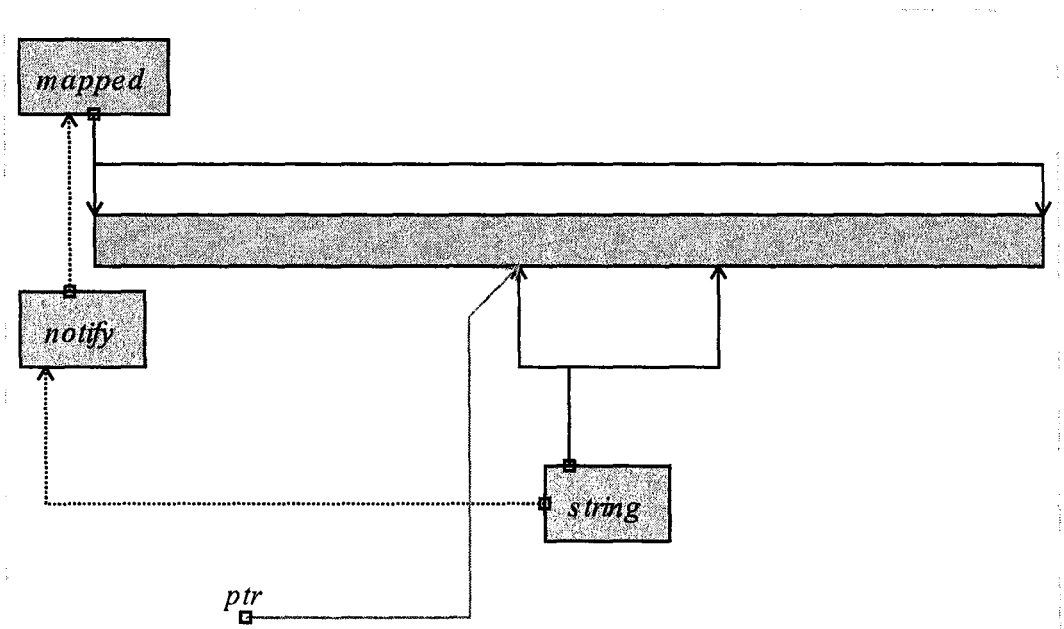

Figure 9.5: a pointer is provided directly to memory-mapped data

At this point, the user is able to use the string directly as if it were a native C string. This will cause the data associated with the string to be paged in (unless it had happened to share a page with its own offsets which we read earlier).

The pointer remains valid for the life of *string*. When *string* is released then it drops its reference on *notify* which will, in turn, call `g_mapped_file_free()` on *mapped*.

```
g_variant_unref (string);
```

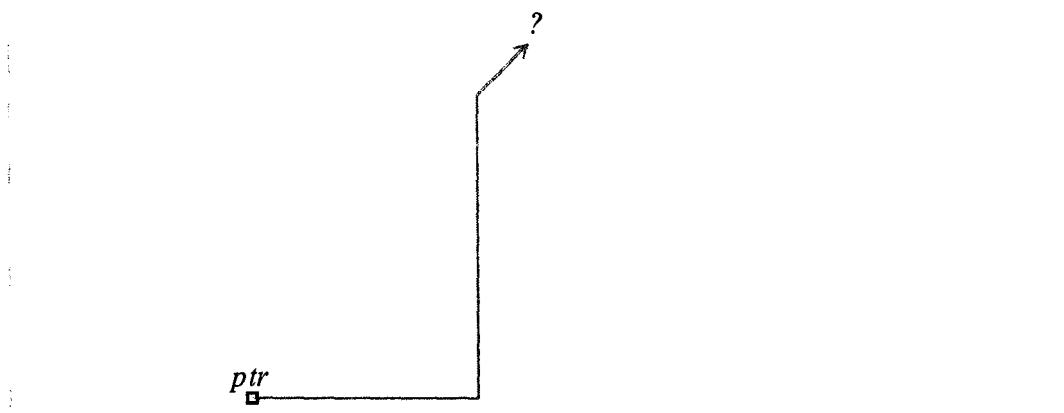

Figure 9.6: the pointer is invalid after the reference is released

Of course, at this point, *ptr* is no longer valid (since *string* has been freed and its serialised data is gone).

9.2 Construction of new values

The second example demonstrates what happens when GVariant is used to construct new values.

Imagine we want to create an array of integers.

```
GVariantBuilder *builder;
GVariant *array;

builder = g_variant_builder_new (G_VARIANT_TYPE_CLASS_ARRAY, NULL);
g_variant_builder_add (builder, "i", 42);
g_variant_builder_add (builder, "i", 28);
g_variant_builder_add (builder, "i", 84);
array = g_variant_builder_end (builder);
```

In this case, a separate `GVariant` instance has been created for each integer. Integer and floating point data is small enough that it can fit inside the `GVariant` structure and need not refer to an external serialised data buffer.

A GVariant instance has also been created to represent the array. This instance contains references to the integers.

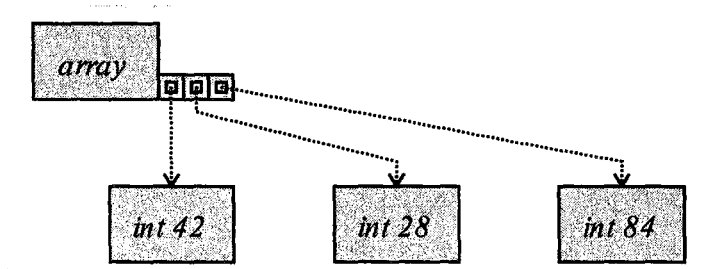

Figure 9.7: a tree of GVariant instances

It may seem wasteful to create an array of integers in this way, but consider the case that we were adding more complicated values to the array. In this case, the cost of copying the data of those values into a separate serialised buffer may outweigh the benefit.

Next, perhaps want to have the array as one of the child items of a structure type. This is easy enough to do.

```
GVariant *structure;

structure = g_variant_new ("(s@ai)", "hello world", array);
```

Note that due to the floating reference counts of GVariant instances the reference held by *array* has been assumed by *structure*.

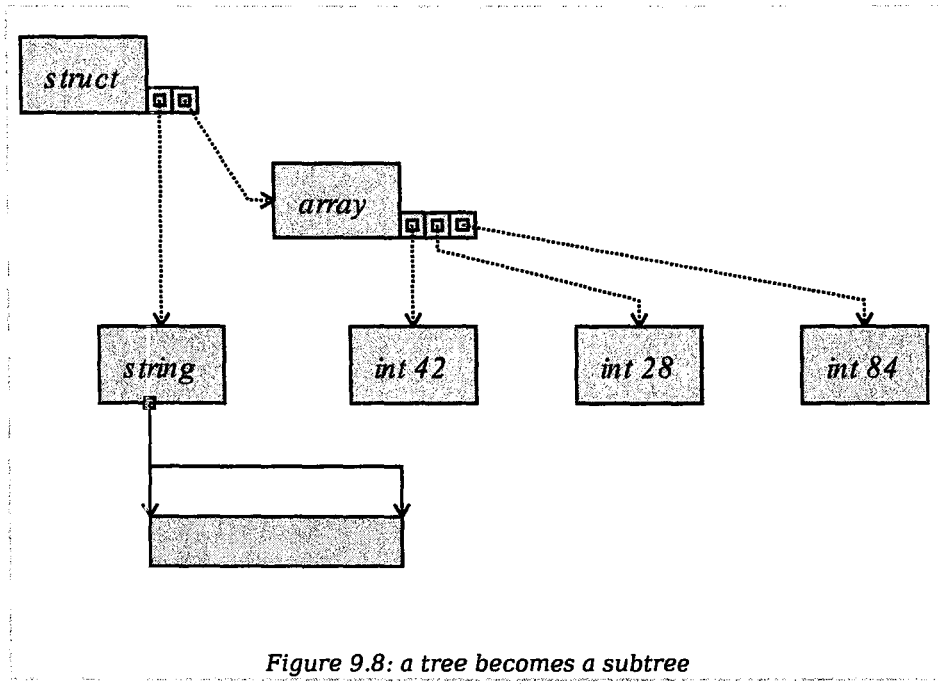

Figure 9.8: a tree becomes a subtree

Nesting can continue in this way to arbitrary depths.

Eventually it may be desirable to access the serialised representation of the value of *structure*. Two different methods are provided for accomplishing this.

First is an API to store the serialised data into a buffer provided by the caller. A call is provided to determine how large this buffer must be.

```
gpointer *data;
gsize size;

size = g_variant_get_size (structure);
data = g_malloc (size);
g_variant_store (structure, data, G_LITTLE_ENDIAN);
```

Each leaf node writes its own data into the buffer and the intermediate container nodes write only the framing information and padding bytes

where appropriate. This means that any given piece of data is written only once and not copied several times as it moves through each layer.

There is also an API to allow access directly to the serialised data of a value.

```
gconstpointer data;
gsize size;

data = g_variant_get_data (structure, &size);
```

This call presents a problem for the case where the value is stored in tree format (as is the case with *structure*). There is no serialised data to return a pointer to.

In order to satisfy the request, a new memory buffer is allocated.

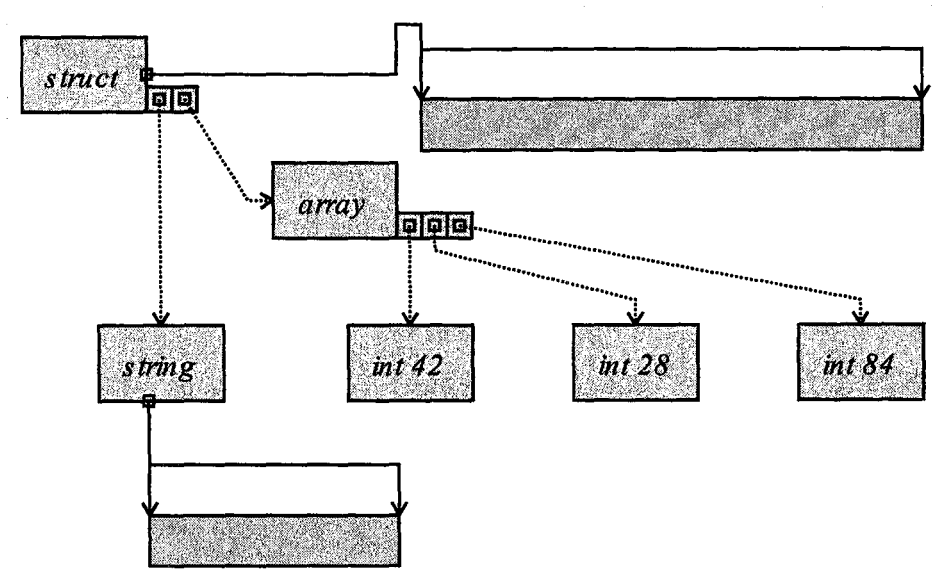

Figure 9.9: implicit serialisation occurs

The serialisation process then occurs with this new buffer as the destination. The process is very similar to the one that occurs when using the first API (specifically, each value stores its own contents directly into the top level buffer).

Once the serialisation is complete, the top level value drops any references it holds on the child values. The instance is no longer in tree form; it has been serialised.

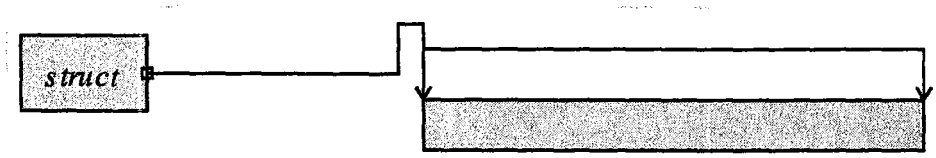

Figure 9.10: after serialisation, the children are released

Any future calls to `g_variant_get_data()` will be able to return immediately. The return value of `g_variant_get_data()` is valid for the life of the instance.

Any future calls to `g_variant_get_child()` will deserialise the child from the serialised data (similar to what occurred in Section 9.1.

Chapter 10

Implementation Details

This chapter presents information about the implementation of GVariant.

10.1 Internal Modularity

Internally, GVariant is separated into a number of separate source files. These separate files are used to separate functionality into logical groupings and to increase modularity by enforcing the principle of loose coupling.

As an example of how loose coupling is forced by this arrangement, all non-trivial structure types in GVariant are declared within a C source file (not a header) and are accessible to only the functions contained in that file. This limits the amount of code that can be affected by a change to one of these structures.

One particular division is worth mentioning because it is not made along boundaries that are implied by logical grouping of functionality. This is the separation between the files `gvariant-util.c` and `gvariant-core.c`.

These files both contain functions that are used for creating and accessing `GVariant` instances. The simple rule for which file a particular function goes in is determined by the fact that the `GVariant` structure is declared in `gvariant-core.c`. If a function requires direct access to the `GVariant` structure then it goes in this file. Everything else goes into `gvariant-util.c`. During development, effort has been made to keep the number of functions in `gvariant-core.c` as small as possible.

10.2 Values

A `GVariant` instance is a small structure type (24 bytes on 32 bit systems) allocated on creation and freed when the last reference to it drops.

Each GVariant instance contains a reference count encoded as an integer. The high bit of this integer is used as the floating reference flag (in order to implement the floating reference behaviour described in Chapter 8.) All reference counting operations are performed using the glib atomic functions and are therefore thread-safe without locking.

Each instance also contains a pointer to a type information structure (described below). The type of the instance never changes, so accessing it is always safe, so long as the instance continues to exist.

Each instance contains a state register, encoded as an integer. The individual bits in this integer value represent various conditions that may or may not be true about the instance. This is explained in considerable detail in Section 10.3.1.

The remainder of the content of the instance is determined by the state that the instance is in (as determined from the state register).

10.3 State Transformations

Initially, the implicit state transitions that a GVariant instance needed to undergo were handled in an ad hoc manner in response to programmer calls.

For example, `g_variant_get_size()` (which reports the byte size of the serialised form of a value) would check if the size was already known (internally: the size field contains a value other than -1) and, if so, report this value directly. Otherwise, if the value was in tree form and, it would call the serialiser to determine the number of bytes that would result from serialising that tree, caching the result.

If another function needed to know this size then, out of interest of avoiding code duplication, it would call `g_variant_get_size()` which would perform the work if necessary (or simply return the cached value if it was available).

With the wide range of state transitions that a `GVariant` instance can undergo, the web of function calls that occurred between these functions was getting difficult to keep track of. Keeping track of when locks needed to be held or not was also becoming difficult. Changes to GVariant would often have unintended side-effects that would only be discovered through unit test failures and a bit of head-scratching.

10.3.1 The Condition Machine

As a method of dealing with this increasingly unmanageable complexity the level of formalism was increased — "conditions" were introduced as the method of dealing with the state of each instance.

A state register was added to the `GVariant` structure. This state register is an integer, that when viewed in binary, has each bit corresponding to a particular condition (for example "size is known"). All hacks about checking if a value was equal to -1 or a pointer was equal to `NULL` were removed.

For any given instance, a condition bit may only ever transition from zero to one; conditions may be false, but they can never change from true to false.

For every defined condition, a transition function was defined to transition the condition from false to true. In the case of the "size known" condition this is the function that would invoke the serialiser and record the result. Each transition function was given a precondition (in terms of other conditions) that had to be satisfied before it could run.

The "condition machine" was developed. Its responsibility is to satisfy requests for certain conditions by executing transition functions to enable them. If the transition function has a precondition then it invokes itself recursively to satisfy that precondition.

Excepting reference counting, only transition functions are allowed to make any change to the state of a `GVariant` instance. Concurrency considerations are greatly simplified; a lock is held whenever the condition machine is operating (including when transition functions are operating), ensuring that no two threads are trying to modify a given instance at a time.

Of course, concurrent read accesses are safe, but we are left to consider the case of concurrent read and write access. This is dealt with by seeing each condition as a sort of promise — if a condition is true then some operation is safe. Because conditions can never be disabled, it is sufficient simply to check that a condition is true before proceeding — no lock required.

There are a small number of operations (all relating to dealing with tree form GVariant instances) that are not safe to perform unless locked (essentially because we need to prevent the tree from disappearing from under us in the case that the value is serialised). The instance is locked in these cases, but the lock is only ever held briefly (typically only for the duration of reading the value of a pointer and increasing a reference count).

Details about the operation of the condition machine and the list of conditions that GVariant uses are given in Appendix C.

10.3.1.1 Comments on the Change

Moving to using the condition machine represented a near-complete redefinition of the `GVariant` structure type. Considerable work was to be expected and significant breakage would be understood.

Due to the separation between `gvariant-core.c` and `gvariant-util.c`, however, the amount of rewriting was kept to a minimum — about two days of work, including development of ideas. This provides some anecdotal evidence for the soundness of the division between these two files.

Providing some evidence for the soundness of giving such explicit treatment to the concept of conditions is the fact that GVariant had a large suite of unit tests that were developed against (and found many bugs in) the old implementation. When the new implementation based on the condition machinery was first written, with the exception of some trivial mistakes, these unit tests all passed right away.

Perhaps a more significant (although somewhat less quantitative) endorsement comes in form of the fact that the author's "clarity" about what's going on has increased considerably. It no longer feels like one wrong move could bring down an entire house of cards.

10.4 Locking

GVariant uses per-instance locking; contention can only occur when separate threads are making use of the same `GVariant` instance.

Plain (non-recursive, no distinction between reader and writer) mutual exclusion locks are used. Since the lock primitives available as part of GLib (`GMutex`, `GStaticMutex`) and POSIX (`pthread_mutex_t`) are all very

large with respect to the size of the GVariant structure, a new mutex lock implementation was developed that requires only a single bit.

This implementation is described in Appendix B.

The highest bit of the state register is used for the lock.

10.5 Type Information

`GVariantTypeInfo` is an opaque structure type that is for internal use only. It is a private implementation detail not exposed to the user.

The purpose of `GVariantTypeInfo` is to act as a cache of information associated with a given `GVariantType`.

10.5.1 Life Cycle

`GVariantTypeInfo` structures come and go as needed, but no more than one exists for a given type. Reference counts are used.

When the type information structure for a given type is required a lookup in a hash table of existing type information is performed. If the type already has an information structure, then that structure has its reference count increased. If the type does not already have an information structure then one is created on the fly and added to the hash table.

When a particular user of the type information is done with the information, they unreference it. If they held the last reference, then currently the type information is destroyed and removed from the table. Future implementation tweaks may involve keeping data cached for a while after it is no longer used.

10.5.2 Information Contained

All type information structures contain some basic information:

Type
the `GVariantType` corresponding to this type information.

Alignment
the alignment requirement for values of this type. Stored as the value that the starting address must be a multiple of, minus 1.

Fixed size
the serialised size that all values of this type share. If the type is not a fixed-size type then 0 is stored (note that all fixed-size types have a non-zero fixed size).

The type is stored as part of the type information structure for two reasons. First, it allows the type to be determined from the type information. This allows GVariant instances to store only a pointer to the type information (and not also to the type). Secondly, the type itself is needed as a key into the hash table when an entry has to be removed.

A couple of derived properties are available from those listed above.

Always native byte order
values of some type are always in native byte order and never need to be byteswapped (strings, for example). Naturally, the items that never need to be byteswapped are exactly those that have no alignment constraints (since they do not contain multi-byte integer values). Knowing this allows for pruning the tree while byteswapping complicated values.

Is container
many operations on `GVariant` instances are only applicable if the value has a container type. This can be easily determined from the type stored here.

For container types, extra type information is stored.

For maybe and array types, only one additional piece of information is stored: a pointer to the element type's `GVariantTypeInfo`. This direct reference means that no string manipulation need be performed to determine this type and no hash table lookup need be performed to find its information structure.

For structure and dictionary entry types as well, the `GVariantTypeInfo` pointers for each item type are stored. Additionally, for each item, information is stored in order to allow constant time lookup of that item within the structure. This information is generated according to the algorithm described in Section 7.2.2 and stored in normalised plus/and/or format (described in Section 7.2.4).

10.6 Serialisation

The serialiser and deserialiser are implemented as three privates interfaces that are used internally by GVariant.

The serialiser only operates on container types. Non-container types have very simple formats and they are accessed directly without additional abstraction.

Determine Size
This call is used to determine the number of bytes that would be required for a buffer to store the serialised form of a `GVariant` instance into. This function is called before serialisation of an instance occurs in order to know how large to make the buffer.

Serialise
This call is used to serialise the value into an existing buffer. The buffer must have already been allocated (using the size returned by the previous operation).

Deserialise

When given a `GVariant` instance, and an index *n*, this call is responsible for determining the sub-sequence of the serialised data of the instance that corresponds to the *n*th child.

This call also consults the `GVariantTypeInfo` structure to determine the type information for the new child instance.

Because all deserialisation operations on the serialisation format can occur in constant time, since the `GVariantTypeInfo` structure contains a direct pointer to all the necessary, and because no data is copied, the deserialisation operation occurs in constant time.

There is one exception (the source of the claim that only "nearly all" deserialisation operations are constant time): when extracting the value from a variant, the type string of the variant must parsed and looked up in the hash table of `GVariantTypeInfo` instances. This operation is linear in the size of the type string.

Chapter 11

Testing

The requirements listed in Chapter 4 are the sorts of requirements that don't easily lend themselves to verification by automated testing. The body of this thesis has been dedicated to providing evidence that these requirements have been satisfied.

There are a whole other class of "obvious" implicit requirements, however, that lend themselves nicely to automated testing.

These requirements are things like "does not crash" and "gives me back the same value I put in". Validation of these requirements has mostly been performed by development of test cases and through early use of GVariant in other projects (see Chapter 13 for more information about these).

This chapter discusses a number of methodologies that were used during the development of the automated tests.

Automated tests were used in two separate ways. Some tests were written after the features that they were meant to test were in place with the intention of finding bugs in the existing software. Other tests were written before the features (with the intention that the tests would immediately fail) in order to drive development. For this reason, it is

hard to make clear statements about exactly how many bugs particular test cases have caught.

11.1 Identity Operations

Looked at from a certain angle, GVariant can be seen as a translation system. It can store data in its native serialisation format, but also allows for the data to be pretty-printed to or parsed from XML. There are also several programmer interfaces to GVariant.

Moving data into GVariant through one of these interfaces and out through another is effectively a translation process. We expect that, over the course of any number of translations, if the value is translated back to its original representation it will be exactly the same as the value that was given in the first place.

Testing this simple property has revealed a number of bugs.

11.2 Random Testing

The number of ways in which different types of GVariant containers can be stacked together is essentially limitless. It can be quite difficult to guess which particular combination might expose flaws in the implementation. Coming up with imaginative test cases is a tiring exercise.

An alternative to manual development of test cases is to write code to produce test cases for you.

For GVariant, such a framework was developed by William Hua. A random well-formed and semantically valid XML document is produced and loaded into GVariant, put through a number of transformations and then printed out again and compared to the original.

Random testing has proven to be extremely successful in discovering a large number of bugs in GVariant.

11.3 Fuzz Testing

Another sort of testing performed is to check how GVariant responds to non-normal serialised data.

This test process is driven by producing a valid serialised byte sequence for a given (randomly generated) value. Random errors are then introduced to this byte sequence such that it is no longer exactly equal to the original. The number of random errors that are introduced is a variable of the test, and several different levels are tried, ranging from 1-bit errors to substantial damage (20% of the bytes randomly replaced).

The "fuzzed" data is then loaded back into GVariant. The introduction of errors could have three possible effects:

- GVariant interprets the serialised data as having the same value as the original, but notices that it is no longer in normal form.

  GVariant interprets the serialised data as having a different value but accepts the data as being in normal form.

  GVariant interprets the serialised data as having a different value and notices that it is no longer in normal form.

Note that it is not possible for the new byte sequence to have the same value while still remaining in normal form because there is only one normal form per value.

In the case that GVariant reports the data to be in normal form, we check to ensure that the value of the data is different than the original value. This testing offers assurance that GVariant will not accept two different serialised byte sequences as normal forms of the same value.

In the case that GVariant reports that the data is not in normal form, the data is normalised and checked to ensure that it differs from the fuzzed data.

This testing method has revealed several bugs, mainly in the validation code. Surprisingly, it also unearthed a couple of bugs in the serialiser.

PART IV

This part contains a summary of the contributions of this work and discusses future work in terms of changes to GVariant itself and projects that intend to use GVariant in substantial ways.

Chapter 12

Summary

In short: it works.

The tests are passing, and people are making use of the work for development of new projects.

GVariant is slated to be included in the next release of GLib which will expose it to a wider range of hackers and more use. Development will continue.

The contributions of this work are the following:

- Collection and description of the best common practices that form the "folk knowledge" of the GNOME community.
- Development of a new serialisation format adhering to these principles.
- Development of a number of techniques to allow constant-time access to data stored in the new serialisation format, including a technique for compiling a table allowing constant-time random access to members of a structure containing variable-width data.

- Development of a working software library allowing creation of and access to data stored in the new serialisation format.

Chapter 13

Future Work

As always, there is still work to be done. This work has been divided into two categories — work on GVariant itself and work on new projects based on GVariant.

13.1 GVariant

Looking forward, the next hurdle for GVariant is to have its API reviewed by the maintainers of GLib and to merge it into this library.

Work is currently underway by Diego Escalante Urrelo to create Python bindings for GVariant. As it turns out, GVariant's type system of arrays and structures maps rather nicely onto Python's type system of lists and tuples.

A number of features have been suggested by users of GVariant. One such feature makes note of the fact that deserialisation of a GVariant instance can occur without having the entire instance in memory, but that this is not true for serialisation. A new builder interface would be added that could stream out to a file or network socket "on the fly" as values are added to it (removing the need to store all of the values in memory at once).

13.2 DBus

Many of the ideas developed for GVariant may eventually find their way into DBus itself.

Considerable discussion has been made about extending DBus's type system in ways described here (and also in ways described by others, such as the addition of a single precision floating point type).

There is also some discussion about changing DBus to use the serialisation format of GVariant in order to avoid having to translate between the two formats.

Work on these ideas is currently blocked only by a shortage of willing contributors.

13.3 GSettings

As the motivating project for this work, GSettings will be one of the first projects to take advantage of GVariant.

Most obviously, GSettings will make use of the serialisation format described here to store values in its settings database.

GSettings will have GVariant as part of its API. Any GVariant value can be provided as the value of a setting to store in the configuration system.

GSettings is a strongly type configuration system based on the concept of schemas. This fits in nicely with GVariant's strong typing. When coupled with GVariant's lack of deserialisation errors and with the `g_variant_new_va()` and `g_variant_get_va()` functions, this allows for a powerful programmer interface.

As a simple example, if a settings schema specifies that the *size* property in the settings database has the type `(ii)` then the following code can be used:

```
int width, height;

g_settings_get (settings, "size", &width, &height);
```

The user can always be sure that they will be left with valid integer values for *width* and *height*.

The current status of the GSettings project is somewhat disrupted due to significant changes that have occurred in GVariant over the past several months. GSettings needs to be brought up to date with these changes before work can continue.

13.4 GBus

Many ideas are planned for a new project — GBus.

The current DBus library was designed as a reference implementation and to allow ease of binding for higher level programming languages. As such, it contains very few facilities that make it convenient to use directly from C. A number of attempts have been made to correct for these shortcomings — such as by the dbus-glib and dbind projects — by binding additional code to the low-level libdbus library.

These projects have had limited success, and still use the libdbus library. Concerns have been raised about the libdbus library in terms of license compatibility with some GNOME projects and attempts to solve these problems have failed due to uncertainty about the ownership of large parts of the DBus code base.

Another problem with the DBus library is that it implements its own linked lists, dynamic strings, hash tables, memory allocation, and many other facilities that are already available in GLib.

GBus is a complete implementation of a new DBus client built using GLib and GVariant. It will consist of a lower level interface (which

will "borrow" some API ideas from dbind) and a higher level interface allowing direct interaction with the GObject object system.

At present, the low level interfaces are partially complete (having been written by the author and by William Hua). It is possible to connect to the bus and send messages, giving GVariant instances as the values for those messages. The API is currently cumbersome and will change. A great deal of work remains.

The high level interfaces are currently in the idea-gathering phase.

13.5 GObject Introspection

The GObject Introspection project plans to provide runtime and statically-accessible information about the functions available on a given GObject. This project is being undertaken by a number of developers — mainly the ones involved in writing language bindings for GNOME.

The introspection information will greatly simplify creating language bindings for new types of GObjects and will also facilitate publishing objects on the message bus for remote procedure calls.

The database containing the lists of functions defined for each object is expected to be collected at compile-time using one of a variety of techniques (code scanning, or by specification in an additional IDL file).

The database will be serialised using the GVariant serialisation format and stored either as a binary blob within the shared library that implements the object or in an additional file accompanying that library.

GVariant will be used to query and access information about particular functions, properties and signals associated with a specific object type.

The main ideas for the GObject Introspection project were gathered this past March. The project is currently under way and making significant progress.

13.6 GVariant Hash File

One particularly simple and useful use of GVariant is to implement a write-once/read-many hash file. These sorts of files are often used to allow fast and memory-efficient access to many smaller files that are often spread out over a number of subdirectories.

Examples in the current desktop include the font and icon caches. GSettings will also have a similar cache for storing schemas.

The use of one large file what can be memory mapped is driven by the desire to have high performance access to these objects and being able to share the memory overhead associated with using them (since the cache file is mapped into dozens of processes as shared memory, the cost to each individual process is low).

Building this type of file using GVariant is a simple exercise. The file is stored as a serialised GVariant value with a type matching `(a*aa(si))` where * matches the type of data to be stored.

Given a list of pairs of key strings and values, a table can be built as follows. Assign each value a number, in sequence. Store those values in an array, in sequence. This forms the `a*` part of the file. Then, choose a prime number to use as the size of the hash table. This becomes the length of the `aa(si)` array. Hash each key string and reduce the result, modulo the prime. Pair each key string with the integer corresponding to its value and store it in the sub-array indexed by the reduced result of the hash.

Lookup is done by hashing the search string with the same hash algorithm and using the reduced result to index into the hash array. Each item in the sub-array is then checked for string equality with the search string. When the correct string is found, the paired integer is used as an index into the first array.

This indirection (not storing the strings and values directly together) is used to keep the number of memory pages used by the hash table as small as possible. The table will be accessed for every single lookup and

mixing the value data in with the table would result in this data being unnecessarily faulted in during table lookups (particularly in the case of traversing highly populated hash chains).

The hash table functionality is generally useful and simple in its implementation so it is likely that it will eventually be included as a core part of GVariant.

Bibliography

[GNOME] **GNOME: The Free Software Desktop Project**
http://www.gnome.org/
last access: 2008-09-29.

[GConf] **GConf configuration system**
http://www.gnome.org/projects/gconf/
last access: 2008-09-29.

[DBus] **D-Bus Specification**
Havoc Pennington. Anders Carlsson. Alexander Larsson.
http://dbus.freedesktop.org/doc/dbus-specification.html
last access: 2008-09-29.

[ast] **Modern Operating Systems**
Second Edition
Andrew S. Tanenbaum.
© 2001 Prentice-Hall, Inc.

[XML] **XML In a Nutshell**
Third Edition
Elliotte Rusty Harold. W. Scott Means.
© 2004 O'Reilly

[CORBA] **CORBA Basics**
Object Management Group.
http://www.omg.org/gettingstarted/corbafaq.htm
last access: 2008-09-29.

[**protobuf**] **Protocol Buffers**
Google.
`http://code.google.com/p/protobuf/`
last access: 2008-09-29.

Appendix A

Interface Reference

This appendix contains a copy of the API reference documentation of GVariant.

The documentation is current and mostly complete at the time of printing. Like any software, however, GVariant is likely to evolve to address future needs and these improvements will cause changes in the interface documented here. As such, this documentation may be out of date.

Updated documentation can be found online.

GVariantTypeClass

Synopsis

```
enum                GVariantTypeClass;

gboolean            g_variant_type_class_is_basic       (GVariantTypeClass class);
gboolean            g_variant_type_class_is_container   (GVariantTypeClass class);
```

Description

Details

enum GVariantTypeClass

```
typedef enum
{
  G_VARIANT_TYPE_CLASS_INVALID          = '\0',
  G_VARIANT_TYPE_CLASS_BOOLEAN           = 'b',
  G_VARIANT_TYPE_CLASS_BYTE              = 'y',

  G_VARIANT_TYPE_CLASS_INT16             = 'n',
  G_VARIANT_TYPE_CLASS_UINT16            = 'q',
  G_VARIANT_TYPE_CLASS_INT32             = 'i',
  G_VARIANT_TYPE_CLASS_UINT32            = 'u',
  G_VARIANT_TYPE_CLASS_INT64             = 'x',
  G_VARIANT_TYPE_CLASS_UINT64            = 't',

  G_VARIANT_TYPE_CLASS_DOUBLE            = 'd',

  G_VARIANT_TYPE_CLASS_STRING            = 's',
  G_VARIANT_TYPE_CLASS_OBJECT_PATH       = 'o',
  G_VARIANT_TYPE_CLASS_SIGNATURE         = 'g',

  G_VARIANT_TYPE_CLASS_VARIANT           = 'v',

  G_VARIANT_TYPE_CLASS_MAYBE             = 'm',
  G_VARIANT_TYPE_CLASS_ARRAY             = 'a',
  G_VARIANT_TYPE_CLASS_STRUCT            = 'r',
  G_VARIANT_TYPE_CLASS_DICT_ENTRY        = 'e',

  G_VARIANT_TYPE_CLASS_ALL               = '*',
  G_VARIANT_TYPE_CLASS_BASIC             = '?'
} GVariantTypeClass;
```

A enumerated type to group GVariantType instances into classes.

If you ever want to perform some sort of recursive operation on the contents of a GVariantType you will probably end up using a switch statement over the GVariantTypeClass of the type and its component sub-types.

A GVariantType is said to "be in" a given GVariantTypeClass. The type classes are overlapping, so a given GVariantType may have more than one type class. For example, G_VARIANT_TYPE_BOOLEAN is of the following classes: G_VARIANT_TYPE_CLASS_BOOLEAN, G_VARIANT_TYPE_CLASS_BASIC, G_VARIANT_TYPE_CLASS_ALL.

G_VARIANT_TYPE_CLASS_INVALID
the class of no type

G_VARIANT_TYPE_CLASS_BOOLEAN
the class containing the type G_VARIANT_TYPE_BOOLEAN

G_VARIANT_TYPE_CLASS_BYTE
the class containing the type G_VARIANT_TYPE_BYTE

G_VARIANT_TYPE_CLASS_INT16
the class containing the type G_VARIANT_TYPE_INT16

G_VARIANT_TYPE_CLASS_UINT16
the class containing the type G_VARIANT_TYPE_UINT16

G_VARIANT_TYPE_CLASS_INT32
the class containing the type G_VARIANT_TYPE_INT32

G_VARIANT_TYPE_CLASS_UINT32
the class containing the type G_VARIANT_TYPE_UINT32

G_VARIANT_TYPE_CLASS_INT64
the class containing the type G_VARIANT_TYPE_INT64

G_VARIANT_TYPE_CLASS_UINT64
the class containing the type G_VARIANT_TYPE_UINT64

G_VARIANT_TYPE_CLASS_DOUBLE
the class containing the type G_VARIANT_TYPE_DOUBLE

G_VARIANT_TYPE_CLASS_STRING
the class containing the type G_VARIANT_TYPE_STRING

G_VARIANT_TYPE_CLASS_OBJECT_PATH
the class containing the type G_VARIANT_TYPE_OBJECT_PATH

G_VARIANT_TYPE_CLASS_SIGNATURE
the class containing the type G_VARIANT_TYPE_SIGNATURE

G_VARIANT_TYPE_CLASS_VARIANT
the class containing the type G_VARIANT_TYPE_VARIANT

G_VARIANT_TYPE_CLASS_MAYBE
the class containing all maybe types

G_VARIANT_TYPE_CLASS_ARRAY
the class containing all array types

G_VARIANT_TYPE_CLASS_STRUCT
the class containing all structure types

G_VARIANT_TYPE_CLASS_DICT_ENTRY
the class containing all dictionary entry types

G_VARIANT_TYPE_CLASS_ALL
the class containing all types (including G_VARIANT_TYPE_ANY and anything that matches it).

G_VARIANT_TYPE_CLASS_BASIC
the class containing all of the basic types (including G_VARIANT_TYPE_ANY_BASIC and anything that matches it).

g_variant_type_class_is_basic ()

```
gboolean
g_variant_type_class_is_basic (GVariantTypeClass class);
```

Determines if *class* is a basic class.

The following are considered to be basic classes: boolean, byte, the signed and unsigned integer classes, double, string, object path and signature. Additionally, the 'basic' type class is also considered to be basic.

class **:**
a GVariantTypeClass

Returns **:**
TRUE if *class* is a basic class

g_variant_type_class_is_container ()

```
gboolean
g_variant_type_class_is_container (GVariantTypeClass class);
```

Determines if *class* is a container class.

The following are considered to be container classes: maybe, array, struct, dict_entry and variant.

class **:**
a GVariantTypeClass

Returns **:**
TRUE if *class* is a container class

GVariantType

Synopsis

```
typedef             GVariantType;

#define             G_VARIANT_TYPE_BOOLEAN
#define             G_VARIANT_TYPE_BYTE
#define             G_VARIANT_TYPE_INT16
#define             G_VARIANT_TYPE_UINT16
#define             G_VARIANT_TYPE_INT32
#define             G_VARIANT_TYPE_UINT32
#define             G_VARIANT_TYPE_INT64
#define             G_VARIANT_TYPE_UINT64
#define             G_VARIANT_TYPE_DOUBLE
#define             G_VARIANT_TYPE_STRING
#define             G_VARIANT_TYPE_OBJECT_PATH
#define             G_VARIANT_TYPE_SIGNATURE
#define             G_VARIANT_TYPE_VARIANT
#define             G_VARIANT_TYPE_ANY
#define             G_VARIANT_TYPE_ANY_BASIC
#define             G_VARIANT_TYPE_ANY_ARRAY
#define             G_VARIANT_TYPE_ANY_DICTIONARY
#define             G_VARIANT_TYPE_ANY_DICT_ENTRY
#define             G_VARIANT_TYPE_ANY_MAYBE
#define             G_VARIANT_TYPE_ANY_STRUCT
#define             G_VARIANT_TYPE_UNIT

gboolean            g_variant_type_string_is_valid      (const gchar *type_string);
gboolean            g_variant_type_string_scan          (const gchar **type_string,
                                                         const gchar *limit);

#define             G_VARIANT_TYPE                      (type_string)
void                g_variant_type_free                 (GVariantType *type);
GVariantType*       g_variant_type_copy                 (const GVariantType *type);
GVariantType*       g_variant_type_new                  (const gchar *type_string);

gsize               g_variant_type_get_string_length    (const GVariantType *type);
const gchar*        g_variant_type_peek_string          (const GVariantType *type);
gchar*              g_variant_type_dup_string           (const GVariantType *type);

GVariantTypeClass   g_variant_type_get_class            (const GVariantType *type);
gboolean            g_variant_type_is_in_class          (const GVariantType *type,
                                                         GVariantTypeClass class);
gboolean            g_variant_type_is_concrete          (const GVariantType *type);
gboolean            g_variant_type_is_container         (const GVariantType *type);
gboolean            g_variant_type_is_basic             (const GVariantType *type);

guint               g_variant_type_hash                 (gconstpointer type);
gboolean            g_variant_type_equal                (gconstpointer type1,
                                                         gconstpointer type2);
gboolean            g_variant_type_matches              (const GVariantType *type,
                                                         const GVariantType *pattern);
```

```
const GVariantType* g_variant_type_element              (const GVariantType *type);
const GVariantType* g_variant_type_first                (const GVariantType *type);
const GVariantType* g_variant_type_next                 (const GVariantType *type);
gsize               g_variant_type_n_items              (const GVariantType *type);
const GVariantType* g_variant_type_key                  (const GVariantType *type);
const GVariantType* g_variant_type_value                (const GVariantType *type);

GVariantType*       g_variant_type_new_maybe            (const GVariantType *element);
GVariantType*       g_variant_type_new_array            (const GVariantType *element);
const GVariantType* (*GVariantTypeGetter)               (gpointer data);
GVariantType*       g_variant_type_new_struct           (gconstpointer *items,
                                                         GVariantTypeGetter func,
                                                         gsize length);
GVariantType*       g_variant_type_new_dict_entry       (const GVariantType *key,
                                                         const GVariantType *value);
```

Description

Details

GVariantType

```
typedef struct OPAQUE_TYPE_GVariantType GVariantType;
```

An opaque type representing either the type of a GVariant instance or a pattern that could match other types.

Each GVariantType has a corresponding type string. The grammar generating all valid type strings is:

```
type = base | 'a' type | 'm' type | 'v' | '*'
     | 'r' | '{' + base + type + '}' | '(' + types + ')'

base = 'b' | 'y' | 'n' | 'q' | 'i' | 'u' | 'x'
     | 't' | 'd' | 's' | 'o' | 'g' | '?'

types = '' | type + types
```

The types that have single character type strings are all defined with their own constants (for example, G_VARIANT_TYPE_BOOLEAN).

The types that have type strings starting with 'a' are array types, where the characters after the 'a' are the type string of the array element type.

The types that have type strings starting with 'm' are maybe types, where the characters after the 'm' are the type string of the maybe element type.

The types that start with '{' and end with '}' are dictionary entry types, where the first contained type string is the one corresponding to the type of the key, and the second is the one corresponding to the type of the value.

The types that start with '(' and end with ')' are structure types, where each type string contained between the brackets corresponds to an item type of that structure type.

Any type that has a type string that can be generated from 'base' is in the `GVariantTypeClass` `G_VARIANT_TYPE_CLASS_BASIC`.

Any type that has a type string that can be generated from 'type' is in the class `G_VARIANT_TYPE_CLASS_ALL`. This is all types.

Each type is a member of exactly one other `GVariantTypeClass`.

Note that, in reality, a `GVariantType` is just a string pointer cast to an opaque type. It is only valid to have a pointer of this type, however, if you are sure that it is a valid type string. Functions that take `GVariantType` as parameters assume that the string is well-formed. Also note that a `GVariantType` is not necessarily nul-terminated.

G_VARIANT_TYPE_BOOLEAN

```
#define G_VARIANT_TYPE_BOOLEAN ((const GVariantType *) "b")
```

The type of a value that can be either TRUE or FALSE.

G_VARIANT_TYPE_BYTE

```
#define G_VARIANT_TYPE_BYTE ((const GVariantType *) "y")
```

The type of an integer value that can range from 0 to 255.

G_VARIANT_TYPE_INT16

```
#define G_VARIANT_TYPE_INT16 ((const GVariantType *) "n")
```

The type of an integer value that can range from -32768 to 32767.

G_VARIANT_TYPE_UINT16

```
#define G_VARIANT_TYPE_UINT16 ((const GVariantType *) "q")
```

The type of an integer value that can range from 0 to 65535. There were about this many people living in Toronto In the 1870s.

G_VARIANT_TYPE_INT32

```
#define G_VARIANT_TYPE_INT32 ((const GVariantType *) "i")
```

The type of an integer value that can range from -2147483648 to 2147483647.

G_VARIANT_TYPE_UINT32

```
#define G_VARIANT_TYPE_UINT32 ((const GVariantType *) "u")
```

The type of an integer value that can range from 0 to 4294967295. That's one number for everyone who was around in the late 1970s.

G_VARIANT_TYPE_INT64

```
#define G_VARIANT_TYPE_INT64 ((const GVariantType *) "x")
```

The type of an integer value that can range from -9223372036854775808 to 9223372036854775807.

G_VARIANT_TYPE_UINT64

```
#define G_VARIANT_TYPE_UINT64 ((const GVariantType *) "t")
```

The type of an integer value that can range from 0 to 18446744073709551616. That's a really big number, but a Rubik's cube can have a bit more than twice as many possible positions.

G_VARIANT_TYPE_DOUBLE

```
#define G_VARIANT_TYPE_DOUBLE ((const GVariantType *) "d")
```

The type of a double precision IEEE754 floating point number. These guys go up to about 1.80e308 (plus and minus) but miss out on some numbers in between. In any case, that's far greater than the estimated number of fundamental particles in the observable universe.

G_VARIANT_TYPE_STRING

```
#define G_VARIANT_TYPE_STRING ((const GVariantType *) "s")
```

The type of a string. "" is a string. NULL is not a string.

G_VARIANT_TYPE_OBJECT_PATH

```
#define G_VARIANT_TYPE_OBJECT_PATH ((const GVariantType *) "o")
```

The type of a DBus object reference. These are strings of a specific format used to identify objects at a given destination on the bus.

G_VARIANT_TYPE_SIGNATURE

```
#define G_VARIANT_TYPE_SIGNATURE ((const GVariantType *) "g")
```

The type of a DBus type signature. These are strings of a specific format used as type signatures for DBus methods and messages.

Any valid GVariantType signature string is a valid DBus type signature. In addition, a concatenation of any number of valid GVariantType signature strings is also a valid DBus type signature.

G_VARIANT_TYPE_VARIANT

```
#define G_VARIANT_TYPE_VARIANT ((const GVariantType *) "v")
```

The type of a box that contains any other value (including another variant).

G_VARIANT_TYPE_ANY

```
#define G_VARIANT_TYPE_ANY ((const GVariantType *) "*")
```

The wildcard type. Matches any type.

G_VARIANT_TYPE_ANY_BASIC

```
#define G_VARIANT_TYPE_ANY_BASIC ((const GVariantType *) "?")
```

A wildcard type matching any basic type.

G_VARIANT_TYPE_ANY_ARRAY

```
#define G_VARIANT_TYPE_ANY_ARRAY ((const GVariantType *) "a*")
```

A wildcard type matching any array type.

G_VARIANT_TYPE_ANY_DICTIONARY

```
#define G_VARIANT_TYPE_ANY_DICTIONARY ((const GVariantType *) "ae")
```

A wildcard type matching any dictionary type.

G_VARIANT_TYPE_ANY_DICT_ENTRY

```
#define G_VARIANT_TYPE_ANY_DICT_ENTRY ((const GVariantType *) "{?*}")
```

A wildcard type matching any dictionary entry type.

G_VARIANT_TYPE_ANY_MAYBE

```
#define G_VARIANT_TYPE_ANY_MAYBE ((const GVariantType *) "m*")
```

A wildcard type matching any maybe type.

G_VARIANT_TYPE_ANY_STRUCT

```
#define G_VARIANT_TYPE_ANY_STRUCT ((const GVariantType *) "r")
```

A wildcard type matching any structure type.

G_VARIANT_TYPE_UNIT

```
#define G_VARIANT_TYPE_UNIT ((const GVariantType *) "()")
```

The empty structure type. Has only one valid instance.

g_variant_type_string_is_valid ()

```
gboolean
g_variant_type_string_is_valid (const gchar *type_string);
```

Checks if *type_string* is a valid GVariantType type string. This call is equivalent to calling g_variant_type_string_scan() and confirming that the following character is a nul terminator.

type_string **:**
a pointer to any string

Returns **:**
TRUE if *type_string* is exactly one valid type string

g_variant_type_string_scan ()

```
gboolean
g_variant_type_string_scan (const gchar **type_string,
                            const gchar *limit);
```

Scan for a single complete and valid GVariantType type string in *type_string*. The memory pointed to by *limit* (or bytes beyond it) is never accessed.

If a valid type string is found, *type_string* is updated to point to the first character past the end of the string that was found and TRUE is returned.

If there is no valid type string starting at *type_string*, or if the type string does not end before *limit* then FALSE is returned and the state of the *type_string* pointer is undefined.

For the simple case of checking if a string is a valid type string, see g_variant_type_string_is_valid().

type_string :
a pointer to any string

limit :
the end of *string*, or NULL

Returns :
TRUE if a valid type string was found

G_VARIANT_TYPE()

```
#define G_VARIANT_TYPE(type_string)
```

Converts a string to a const GVariantType. Depending on the current debugging level, this function may perform a runtime check to ensure that *string* is valid.

It is always a programmer error to use this macro with an invalid type string.

type_string :
a well-formed GVariantType type string

g_variant_type_free ()

```
void
g_variant_type_free (GVariantType *type);
```

Frees a GVariantType that was allocated with g_variant_type_copy(), g_variant_type_new() or one of the container type constructor functions.

type :
a GVariantType

g_variant_type_copy ()

```
GVariantType *
g_variant_type_copy (const GVariantType *type);
```

Makes a copy of a GVariantType. This copy must be freed using g_variant_type_free().

type :
a GVariantType

Returns :
a new GVariantType

g_variant_type_new ()

```
GVariantType *
g_variant_type_new (const gchar *type_string);
```

Creates a new GVariantType corresponding to the type string given by *type_string*. This new type must be freed using g_variant_type_free().

It is an error to call this function with an invalid type string.

type_string :
a valid GVariantType type string

Returns **:**
a new GVariantType

g_variant_type_get_string_length ()

```
gsize
g_variant_type_get_string_length (const GVariantType *type);
```

Returns the length of the type string corresponding to the given *type*. This function must be used to determine the valid extent of the memory region returned by g_variant_type_peek_string().

type **:**
a GVariantType

Returns **:**
the length of the corresponding type string

g_variant_type_peek_string ()

```
const gchar *
g_variant_type_peek_string (const GVariantType *type);
```

Returns the type string corresponding to the given *type*. The result is not nul-terminated; in order to determine its length you must call g_variant_type_get_string_length().

To get a nul-terminated string, see g_variant_type_dup_string().

type **:**
a GVariantType

Returns **:**
the corresponding type string (non-terminated)

g_variant_type_dup_string ()

```
gchar *
g_variant_type_dup_string (const GVariantType *type);
```

Returns a newly-allocated copy of the type string corresponding to *type*. The return result must be freed using `g_free()`.

type **:**
a `GVariantType`

Returns **:**
the corresponding type string (must be freed)

g_variant_type_get_class ()

```
GVariantTypeClass
g_variant_type_get_class (const GVariantType *type);
```

Determines the smallest type class containing *type*.

For example, although `G_VARIANT_TYPE_CLASS_ALL` matches all types, it will never be returned by this function except for the type `G_VARIANT_TYPE_ANY`.

type **:**
a `GVariantType`

Returns **:**
the smallest class containing *type*

g_variant_type_is_in_class ()

```
gboolean
g_variant_type_is_in_class (const GVariantType *type,
                            GVariantTypeClass class);
```

Determines if *type* is contained within *class*.

Note that the class `G_VARIANT_TYPE_CLASS_ALL` contains every type and the class `G_VARIANT_TYPE_CLASS_BASIC` contains every basic type.

type :
a GVariantType

class :
a GVariantTypeClass

Returns :
TRUE if *type* is in the given *class*

g_variant_type_is_concrete ()

```
gboolean
g_variant_type_is_concrete (const GVariantType *type);
```

Determines if the given *type* is a concrete (ie: non-wildcard) type. A GVariant instance may only have a concrete type.

A type is concrete if its type string does not contain any wildcard characters ('*', '?' or 'r').

type :
a GVariantType

Returns :
TRUE if *type* is concrete

g_variant_type_is_container ()

```
gboolean
g_variant_type_is_container (const GVariantType *type);
```

Determines if the given *type* is a container type.

Container types are any array, maybe, structure, or dictionary entry types plus the variant type.

This function returns TRUE for any wildcard type for which every matching concrete type is a container. This does not include G_VARIANT_TYPE_ANY.

type :
a GVariantType

Returns :
TRUE if *type* is a container type

g_variant_type_is_basic ()

```
gboolean
g_variant_type_is_basic (const GVariantType *type);
```

Determines if the given *type* is a basic type.

Basic types are booleans, bytes, integers, doubles, strings, object paths and signatures.

Only a basic type may be used as the key of a dictionary entry.

This function returns FALSE for all wildcard types except G_VARIANT_TYPE_ANY_BASIC.

type :
a GVariantType

Returns :
TRUE if *type* is a basic type

g_variant_type_hash ()

```
guint
g_variant_type_hash (gconstpointer type);
```

Hashes *type*.

The argument type of *type* is only gconstpointer to allow use with GHashTable without function pointer casting. A valid GVariantType must be provided.

type :
a GVariantType

Returns :
the hash value

g_variant_type_equal ()

```
gboolean
g_variant_type_equal (gconstpointer type1,
                      gconstpointer type2);
```

Compares *type1* and *type2* for equality.

Only returns TRUE if the types are exactly equal. Even if one type is a wildcard type and the other matches it, false will be returned if they are not exactly equal. If you want to check for matching, use g_variant_type_matches().

The argument types of *type1* and *type2* are only gconstpointer to allow use with GHashTable without function pointer casting. For both arguments, a valid GVariantType must be provided.

type1 :
a GVariantType

type2 :
a GVariantType

Returns :
TRUE if *type1* and *type2* are exactly equal

g_variant_type_matches ()

```
gboolean
g_variant_type_matches (const GVariantType *type,
                        const GVariantType *pattern);
```

Performs a pattern match between *type* and *pattern*.

This function returns TRUE if *type* can be reached by making *pattern* less general (ie: by replacing zero or more wildcard characters in the type string of *pattern* with matching type strings that possibly contain wildcards themselves).

This function defines a bounded join-semilattice over GVariantType for which G_VARIANT_TYPE_ANY is top.

type **:**
a GVariantType

pattern **:**
a GVariantType

Returns **:**
TRUE if *type* matches *pattern*

g_variant_type_element ()

```
const GVariantType *
g_variant_type_element (const GVariantType *type);
```

Determines the element type of an array or maybe type.

This function must be called with a type in one of the classes G_VARIANT_TYPE_CLASS_MAYBE or G_VARIANT_TYPE_CLASS_ARRAY.

type **:**
a GVariantType of class array or maybe

Returns **:**
the element type of *type*

g_variant_type_first ()

```
const GVariantType *
g_variant_type_first (const GVariantType *type);
```

Determines the first item type of a structure or dictionary entry type.

This function must be called with a type in one of the classes G_VARIANT_TYPE_CLASS_STRUCT or G_VARIANT_TYPE_CLASS_DICT_ENTRY but must not be called on the generic structure type G_VARIANT_TYPE_ANY_STRUCT.

In the case of a dictionary entry type, this returns the type of the key.

NULL is returned in case of *type* being G_VARIANT_TYPE_UNIT.

This call, together with g_variant_type_next() provides an iterator interface over structure and dictionary entry types.

type **:**
a GVariantType of class struct or dict entry

Returns **:**
the first item type of *type*, or NULL

g_variant_type_next ()

```
const GVariantType *
g_variant_type_next (const GVariantType *type);
```

Determines the next item type of a structure or dictionary entry type.

type must be the result of a previous call to g_variant_type_first(). Together, these two functions provide an iterator interface over structure and dictioanry entry types.

If called on the key type of a dictionary entry then this call returns the value type.

NULL is returned when *type* is the last item in a structure or the value type of a dictionary entry.

type **:**
a GVariantType

Returns **:**
the next GVariantType after *type*, or NULL

g_variant_type_n_items ()

```
gsize
g_variant_type_n_items (const GVariantType *type);
```

Determines the number of items contained in a structure or dictionary entry type.

This function must be called with a type in one of the classes G_VARIANT_TYPE_CLASS_STRUCT or G_VARIANT_TYPE_CLASS_DICT_ENTRY but must not be called on the generic structure type G_VARIANT_TYPE_ANY_STRUCT.

In the case of a dictionary entry type, this function will always return 2.

type **:**
a GVariantType of class struct or dict entry

Returns **:**
the number of items in *type*

g_variant_type_key ()

```
const GVariantType *
g_variant_type_key (const GVariantType *type);
```

Determines the key type of a dictionary entry type.

This function must be called with a type in the class G_VARIANT_TYPE_CLASS_DICT_ENTRY. Other than that, this call is exactly equivalent to g_variant_type_first().

type **:**
a GVariantType of class dict entry

Returns :
the key type of the dictionary entry

g_variant_type_value ()

```
const GVariantType *
g_variant_type_value (const GVariantType *type);
```

Determines the value type of a dictionary entry type.

This function must be called with a type in the class G_VARIANT_TYPE_CLASS_DICT_ENTRY.

type :
a GVariantType of class dict entry

Returns :
the value type of the dictionary entry

g_variant_type_new_maybe ()

```
GVariantType *
g_variant_type_new_maybe (const GVariantType *element);
```

Constructs the type corresponding to a maybe instance containing type *type*.

The result of this function must be freed with a call to g_variant_type_free().

element :
a GVariantType

Returns :
a new maybe GVariantType

g_variant_type_new_array ()

```
GVariantType *
g_variant_type_new_array (const GVariantType *element);
```

Constructs the type corresponding to an array of elements of the type *type*.

The result of this function must be freed with a call to `g_variant_type_free()`.

element **:**
a `GVariantType`

Returns **:**
a new array `GVariantType`

`GVariantTypeGetter ()`

```
constGVariantType* (*GVariantTypeGetter)            (gpointer data);
```

A callback function intended for use with `g_variant_type_new_struct()`. This function's purpose is to extract a `GVariantType` from some pointer type. The returned type should be owned by whatever is at the end of the pointer because it won't be freed.

data **:**
a pointer

Returns **:**
a const `GVariantType`

`g_variant_type_new_struct ()`

```
GVariantType *
g_variant_type_new_struct (gconstpointer *items,
                           GVariantTypeGetter func,
                           gsize length);
```

Constructs a new structure type.

The item types for the structure type may be provided directly (as an array of GVariantType), in which case *func* should be NULL.

The item types can also be provided indirectly. In this case, *items* should be an array of pointers which are passed one at a time to *func* to determine the corresponding GVariantType. For example, you might provide an array of GVariant pointers for *items* and g_variant_get_type() for *func*.

The result of this function must be freed with a call to g_variant_type_free().

items :
an array of items, one for each item

func :
a function to determine each item type

length :
the length of *items*

Returns :
a new GVariantType

g_variant_type_new_dict_entry ()

```
GVariantType *
g_variant_type_new_dict_entry (const GVariantType *key,
                               const GVariantType *value);
```

Constructs the type corresponding to a dictionary entry with a key of type *key* and a value of type *value*.

The result of this function must be freed with a call to g_variant_type_free().

key :
a basic GVariantType

value **:**
a `GVariantType`

Returns **:**
a new dictionary entry `GVariantType`

GVariant

Synopsis

```
typedef             GVariant;
GVariant*           g_variant_new                   (const gchar *format_string,
                                                     ...);
void                g_variant_get                   (GVariant *value,
                                                     const gchar *format_string,
                                                     ...);

GVariant*           g_variant_ref                   (GVariant *value);
GVariant*           g_variant_ref_sink              (GVariant *value);
void                g_variant_unref                 (GVariant *value);
void                g_variant_flatten               (GVariant *value);

GVariantTypeClass   g_variant_get_type_class        (GVariant *value);
const GVariantType* g_variant_get_type              (GVariant *value);
const gchar*        g_variant_get_type_string       (GVariant *value);
gboolean            g_variant_is_basic              (GVariant *value);
gboolean            g_variant_is_container          (GVariant *value);
gboolean            g_variant_matches               (GVariant *value,
                                                     const GVariantType *pattern);

gboolean            g_variant_format_string_scan    (const gchar **format_string);
GVariant*           g_variant_new_va                (const gchar **format_string,
                                                     va_list *app);
void                g_variant_get_va                (GVariant *value,
                                                     const gchar **format_string,
                                                     va_list *app);

GVariant*           g_variant_new_boolean           (gboolean boolean);
GVariant*           g_variant_new_byte              (guint8 byte);
GVariant*           g_variant_new_uint16            (guint16 uint16);
GVariant*           g_variant_new_int16             (gint16 int16);
GVariant*           g_variant_new_uint32            (guint32 uint32);
GVariant*           g_variant_new_int32             (gint32 int32);
GVariant*           g_variant_new_uint64            (guint64 uint64);
GVariant*           g_variant_new_int64             (gint64 int64);
GVariant*           g_variant_new_double            (gdouble floating);
GVariant*           g_variant_new_string            (const gchar *string);
GVariant*           g_variant_new_object_path       (const gchar *string);
gboolean            g_variant_is_object_path        (const gchar *string);
GVariant*           g_variant_new_signature         (const gchar *string);
gboolean            g_variant_is_signature          (const gchar *string);
GVariant*           g_variant_new_variant           (GVariant *value);

gboolean            g_variant_get_boolean           (GVariant *value);
guint8              g_variant_get_byte              (GVariant *value);
guint16             g_variant_get_uint16            (GVariant *value);
gint16              g_variant_get_int16             (GVariant *value);
guint32             g_variant_get_uint32            (GVariant *value);
gint32              g_variant_get_int32             (GVariant *value);
```

```
guint64                 g_variant_get_uint64                (GVariant *value);
gint64                  g_variant_get_int64                 (GVariant *value);
gdouble                 g_variant_get_double                (GVariant *value);
const gchar*            g_variant_get_string                (GVariant *value,
                                                             gsize *length);
gchar*                  g_variant_dup_string                (GVariant *value,
                                                             gsize *length);
GVariant*               g_variant_get_variant               (GVariant *value);

gsize                   g_variant_n_children                (GVariant *value);
GVariant*               g_variant_get_child                 (GVariant *value,
                                                             gsize index);
gconstpointer           g_variant_get_fixed                 (GVariant *value,
                                                             gsize size);
gconstpointer           g_variant_get_fixed_array           (GVariant *value,
                                                             gsize elem_size,
                                                             gsize *length);

typedef                 GVariantIter;
gsize                   g_variant_iter_init                 (GVariantIter *iter,
                                                             GVariant *value);
GVariant*               g_variant_iter_next                 (GVariantIter *iter);
void                    g_variant_iter_cancel               (GVariantIter *iter);
gboolean                g_variant_iter_was_cancelled        (GVariantIter *iter);
gboolean                g_variant_iterate                   (GVariantIter *iter,
                                                             const gchar *format_string,
                                                             ...);

typedef                 GVariantBuilder;
#define                 G_VARIANT_BUILDER_ERROR
enum                    GVariantBuilderError;
void                    g_variant_builder_cancel            (GVariantBuilder *builder);
void                    g_variant_builder_add               (GVariantBuilder *builder,
                                                             const gchar *format_string,
                                                             ...);
void                    g_variant_builder_add_value         (GVariantBuilder *builder,
                                                             GVariant *value);
gboolean                g_variant_builder_check_add         (GVariantBuilder *builder,
                                                             GVariantTypeClass class,
                                                             const GVariantType *type,
                                                             GError **error);
gboolean                g_variant_builder_check_end         (GVariantBuilder *builder,
                                                             GError **error);
GVariantBuilder*        g_variant_builder_close             (GVariantBuilder *child);
GVariant*               g_variant_builder_end               (GVariantBuilder *builder);
GVariantBuilder*        g_variant_builder_new               (GVariantTypeClass class,
                                                             const GVariantType *type);
GVariantBuilder*        g_variant_builder_open              (GVariantBuilder *parent,
                                                             GVariantTypeClass class,
                                                             const GVariantType *type);

GString*                g_variant_markup_print              (GVariant *value,
                                                             GString *string,
                                                             gboolean newlines,
                                                             gint indentation,
                                                             gint tabstop);
```

```
GVariant*           g_variant_markup_parse             (const gchar *text,
                                                        gssize text_len,
                                                        const GVariantType *type,
                                                        GError **error);
void                g_variant_markup_subparser_start   (GMarkupParseContext *context,
                                                        const GVariantType *type);
GVariant*           g_variant_markup_subparser_end     (GMarkupParseContext *context,
                                                        GError **error);
GMarkupParseContext* g_variant_markup_parse_context_new (GMarkupParseFlags flags,
                                                        const GVariantType *type);
GVariant*           g_variant_markup_parse_context_end (GMarkupParseContext *context,
                                                        GError **error);
```

Description

Details

GVariant

```
typedef struct OPAQUE_TYPE__GVariant GVariant;
```

GVariant is an opaque data structure and can only be accessed using the following functions.

g_variant_new ()

```
GVariant *
g_variant_new (const gchar *format_string,
               ...);
```

Creates a new GVariant instance.

Think of this function as an analogue to g_strdup_printf().

The type of the created instance and the arguments that are expected by this function are determined by *format_string*. In the most simple case, *format_string* is exactly equal to a concrete GVariantType type string and the result is of that type. All exceptions to this case are explicitly mentioned below.

The arguments that this function collects are determined by scanning *format_string* from start to end. Brackets do not impact the collection of arguments. Each other character that is encountered will result in an argument being collected.

Arguments for the base types are expected as follows:

If a 'v' character is encountered in *format_string* then a (GVariant *) is collected which must be non-NULL and must point to a valid GVariant instance.

If an array type is encountered in *format_string*, a GVariantBuilder is collected and has g_variant_builder_end() called on it. The type of the array has no impact on argument collection but is checked against the type of the array and can be used to infer the type of an empty array.

If a maybe type is encountered in *format_string*, then the expected arguments vary depending on the type.

If a '*' character is encountered in *format_string* then a (GVariant *) is collected which must be non-NULL and must point to a valid GVariant instance. This GVariant is inserted directly at the given position.

Please note that the syntax of the format string is very likely to be extended in the future.

format_string **:**
a GVariant format string

... **:**
arguments, as per *format_string*

Returns **:**
a new floating GVariant instance

g_variant_get ()

```
void
g_variant_get (GVariant *value,
               const gchar *format_string,
               ...);
```

value :

format_string :

... :

g_variant_ref ()

```
GVariant *
g_variant_ref (GVariant *value);
```

Increases the reference count of *variant*.

value :
a GVariant

Returns :
the same *variant*

g_variant_ref_sink ()

```
GVariant *
g_variant_ref_sink (GVariant *value);
```

If *value* is floating, mark it as no longer floating. If it is not floating, increase its reference count.

value :
a GVariant

Returns :
the same *variant*

g_variant_unref ()

```
void
g_variant_unref (GVariant *value);
```

Decreases the reference count of *variant*. When its reference count drops to 0, the memory used by the variant is freed.

value **:**
a GVariant

g_variant_flatten ()

```
void
g_variant_flatten (GVariant *value);
```

Flattens *value*.

This is a strange function with no direct effects but some noteworthy side-effects. Essentially, it ensures that *value* is in its most favourable form. This involves ensuring that the value is serialised and in machine byte order. The investment of time now can pay off by allowing shorter access times for future calls and typically results in a reduction of memory consumption.

A value received over the network or read from the disk in machine byte order is already flattened.

Some of the effects of this call are that any future accesses to the data of *value* (or children taken from it after flattening) will occur in O(1) time. Also, any data accessed from such a child value will continue to be valid even after the child has been destroyed, as long as *value* still exists (since the contents of the children are now serialised as part of the parent).

value **:**
a GVariant instance

g_variant_get_type_class ()

```
GVariantTypeClass
g_variant_get_type_class (GVariant *value);
```

Returns the type class of *value*. This function is equivalent to calling `g_variant_get_type()` followed by `g_variant_type_get_class()`.

value **:**
a `GVariant`

Returns **:**
the `GVariantTypeClass` of *value*

g_variant_get_type ()

```
const GVariantType *
g_variant_get_type (GVariant *value);
```

Determines the type of *value*.

The return value is valid for the lifetime of *value* and must not be freed.

value **:**
a `GVariant`

Returns **:**
a `GVariantType`

g_variant_get_type_string ()

```
const gchar *
g_variant_get_type_string (GVariant *value);
```

Returns the type string of *value*. Unlike the result of calling `g_variant_type_peek_string()`, this string is nul-terminated. This string belongs to `GVariant` and must not be freed.

value **:**
a `GVariant`

Returns **:**
the type string for the type of *value*

g_variant_is_basic ()

```
gboolean
g_variant_is_basic (GVariant *value);
```

Determines if *value* has a basic type. Values with basic types may be used as the keys of dictionary entries.

This function is the exact opposite of `g_variant_is_container()`.

value **:**
a `GVariant`

Returns **:**
TRUE if *value* has a basic type

g_variant_is_container ()

```
gboolean
g_variant_is_container (GVariant *value);
```

Determines if *value* has a container type. Values with container types may be used with the functions `g_variant_n_children()` and `g_variant_get_child()`.

This function is the exact opposite of `g_variant_is_basic()`.

value **:**
a `GVariant`

Returns **:**
TRUE if *value* has a basic type

g_variant_matches ()

```
gboolean
g_variant_matches (GVariant *value,
                   const GVariantType *pattern);
```

Checks if a value has a type matching the provided pattern. This call is equivalent to calling g_variant_get_type() then g_variant_type_matches().

value **:**
a GVariant instance

pattern **:**
a GVariantType

Returns **:**
TRUE if the type of *value* matches *pattern*

g_variant_format_string_scan ()

```
gboolean
g_variant_format_string_scan (const gchar **format_string);
```

Checks the string pointed to by *format_string* for starting with a properly formed GVariant varargs format string. If a format string is fount, *format_string* is updated to point to the first character following the format string and TRUE is returned.

If no valid format string is found, FALSE is returned and the state of the *format_string* pointer is undefined.

All valid GVariantType strings are also valid format strings. See g_variant_type_string_is_valid().

Additionally, any type string contained in the format string may be prefixed with a '@' character. Nested '@' characters may not appear.

Additionally, any fixed-width type may be prefixed with a '&' character. No wildcard type is a fixed-width type. Like '@', '&' characters may not be nested.

No '@' or '&' character, however, may appear as part of an array type.

Currently, there are no other permissible format strings. Others may be added in the future.

For an explanation of what these strings mean, see `g_variant_new()` and `g_variant_get()`.

format_string **:**
a pass-by-reference pointer to the start of a possible format string

Returns **:**
TRUE if a format string was scanned

g_variant_new_va ()

```
GVariant *
g_variant_new_va (const gchar **format_string,
                  va_list *app);
```

This function is intended to be used by libraries based on `GVariant` that want to provide `g_variant_new()`-like functionality to their users.

The API is more general than `g_variant_new()` to allow a wider range of possible uses.

format_string must still point to a valid format string, but it need not be nul-terminated. Instead, *format_string* is updated to point to the first character past the end of the given format string.

app is a pointer to a `va_list`. The arguments, according to *format_string*, are collected from this `va_list` and the list is left pointing to the argument following the last.

These two generalisations allow mixing of multiple calls to g_variant_new_va() and g_variant_get_va() within a single actual varargs call by the user.

format_string :
a pointer to a format string

app :
a pointer to a va_list

Returns :
a new, floating, GVariant

g_variant_get_va ()

```
void
g_variant_get_va (GVariant *value,
                  const gchar **format_string,
                  va_list *app);
```

This function is intended to be used by libraries based on GVariant that want to provide g_variant_new()-like functionality to their users.

The API is more general than g_variant_get() to allow a wider range of possible uses.

format_string must still point to a valid format string, but it need not be nul-terminated. Instead, *format_string* is updated to point to the first character past the end of the given format string.

app is a pointer to a va_list. The arguments, according to *format_string*, are collected from this va_list and the list is left pointing to the argument following the last.

These two generalisations allow mixing of multiple calls to g_variant_new_va() and g_variant_get_va() within a single actual varargs call by the user.

value :
a GVariant

format_string :
a pointer to a format string

app :
a pointer to a va_list

g_variant_new_boolean ()

```
GVariant *
g_variant_new_boolean (gboolean boolean);
```

Creates a new boolean GVariant instance -- either TRUE or FALSE.

boolean :
a gboolean value

Returns :
a new boolean GVariant instance

g_variant_new_byte ()

```
GVariant *
g_variant_new_byte (guint8 byte);
```

Creates a new byte GVariant instance.

byte :
a guint8 value

Returns :
a new byte GVariant instance

g_variant_new_uint16 ()

```
GVariant *
g_variant_new_uint16 (guint16 uint16);
```

Creates a new uint16 GVariant instance.

uint16 :
a guint16 value

Returns :
a new byte GVariant instance

g_variant_new_int16 ()

```
GVariant *
g_variant_new_int16 (gint16 int16);
```

Creates a new int16 GVariant instance.

int16 :
a gint16 value

Returns :
a new byte GVariant instance

g_variant_new_uint32 ()

```
GVariant *
g_variant_new_uint32 (guint32 uint32);
```

Creates a new uint32 GVariant instance.

uint32 :
a guint32 value

Returns :
a new uint32 GVariant instance

g_variant_new_int32 ()

```
GVariant *
g_variant_new_int32 (gint32 int32);
```

Creates a new int32 GVariant instance.

int32 :
a gint32 value

Returns :
a new byte GVariant instance

g_variant_new_uint64 ()

```
GVariant *
g_variant_new_uint64 (guint64 uint64);
```

Creates a new uint64 GVariant instance.

uint64 :
a guint64 value

Returns :
a new uint64 GVariant instance

g_variant_new_int64 ()

```
GVariant *
g_variant_new_int64 (gint64 int64);
```

Creates a new int64 GVariant instance.

int64 :
a gint64 value

Returns :
a new byte GVariant instance

g_variant_new_double ()

```
GVariant *
g_variant_new_double (gdouble floating);
```

Creates a new double GVariant instance.

floating :
a gdouble floating point value

Returns :
a new double GVariant instance

g_variant_new_string ()

```
GVariant *
g_variant_new_string (const gchar *string);
```

Creates a string GVariant with the contents of *string*.

string :
a normal C nul-terminated string

Returns :
a new string GVariant instance

g_variant_new_object_path ()

```
GVariant *
g_variant_new_object_path (const gchar *string);
```

Creates a DBus object path GVariant with the contents of *string*. *string* must be a valid DBus object path. Use g_variant_is_object_path() if you're not sure.

string :
a normal C nul-terminated string

Returns :
a new object path GVariant instance

g_variant_is_object_path ()

```
gboolean
g_variant_is_object_path (const gchar *string);
```

Determines if a given string is a valid DBus object path. You should ensure that a string is a valid DBus object path before passing it to `g_variant_new_object_path()`.

A valid object path starts with '/' followed by zero or more sequences of characters separated by '/' characters. Each sequence must contain only the characters "[A-Z][a-z][0-9]_". No sequence (including the one following the final '/' character) may be empty.

string **:**
a normal C nul-terminated string

Returns **:**
TRUE if *string* is a DBus object path

g_variant_new_signature ()

```
GVariant *
g_variant_new_signature (const gchar *string);
```

Creates a DBus type signature `GVariant` with the contents of *string*. *string* must be a valid DBus type signature. Use `g_variant_is_signature()` if you're not sure.

string **:**
a normal C nul-terminated string

Returns **:**
a new signature `GVariant` instance

g_variant_is_signature ()

```
gboolean
g_variant_is_signature (const gchar *string);
```

Determines if a given string is a valid DBus type signature. You should ensure that a string is a valid DBus object path before passing it to `g_variant_new_signature()`.

DBus type signatures consist of zero or more concrete `GVariantType` strings in sequence.

string **:**
a normal C nul-terminated string

Returns **:**
TRUE if *string* is a DBus type signature

g_variant_new_variant ()

```
GVariant *
g_variant_new_variant (GVariant *value);
```

Boxes *value*. The result is a `GVariant` instance representing a variant containing the original value.

value **:**
a `GVariance` instance

Returns **:**
a new variant `GVariant` instance

g_variant_get_boolean ()

```
gboolean
g_variant_get_boolean (GVariant *value);
```

Returns the boolean value of *value*.

It is an error to call this function with a *value* of any type other than `G_VARIANT_TYPE_BOOLEAN`.

value :
a boolean GVariant instance

Returns :
TRUE or FALSE

g_variant_get_byte ()

```
guint8
g_variant_get_byte (GVariant *value);
```

Returns the byte value of *value*.

It is an error to call this function with a *value* of any type other than G_VARIANT_TYPE_BYTE.

value :
a byte GVariant instance

Returns :
a guchar

g_variant_get_uint16 ()

```
guint16
g_variant_get_uint16 (GVariant *value);
```

Returns the 16-bit unsigned integer value of *value*.

It is an error to call this function with a *value* of any type other than G_VARIANT_TYPE_UINT16.

value :
a uint16 GVariant instance

Returns :
a guint16

g_variant_get_int16 ()

```
gint16
g_variant_get_int16 (GVariant *value);
```

Returns the 16-bit signed integer value of *value*.

It is an error to call this function with a *value* of any type other than G_VARIANT_TYPE_INT16.

value **:**
a int16 GVariant instance

Returns **:**
a gint16

g_variant_get_uint32 ()

```
guint32
g_variant_get_uint32 (GVariant *value);
```

Returns the 32-bit unsigned integer value of *value*.

It is an error to call this function with a *value* of any type other than G_VARIANT_TYPE_UINT32.

value **:**
a uint32 GVariant instance

Returns **:**
a guint32

g_variant_get_int32 ()

```
gint32
g_variant_get_int32 (GVariant *value);
```

Returns the 32-bit signed integer value of *value*.

It is an error to call this function with a *value* of any type other than G_VARIANT_TYPE_INT32.

value **:**
a int32 GVariant instance

Returns **:**
a gint32

g_variant_get_uint64 ()

```
guint64
g_variant_get_uint64 (GVariant *value);
```

Returns the 64-bit unsigned integer value of *value*.

It is an error to call this function with a *value* of any type other than G_VARIANT_TYPE_UINT64.

value **:**
a uint64 GVariant instance

Returns **:**
a guint64

g_variant_get_int64 ()

```
gint64
g_variant_get_int64 (GVariant *value);
```

Returns the 64-bit signed integer value of *value*.

It is an error to call this function with a *value* of any type other than G_VARIANT_TYPE_INT64.

value **:**
a int64 GVariant instance

Returns **:**
a gint64

g_variant_get_double ()

```
gdouble
g_variant_get_double (GVariant *value);
```

Returns the double precision floating point value of *value*.

It is an error to call this function with a *value* of any type other than G_VARIANT_TYPE_DOUBLE.

value **:**
a double GVariant instance

Returns **:**
a gdouble

g_variant_get_string ()

```
const gchar *
g_variant_get_string (GVariant *value,
                      gsize *length);
```

Returns the string value of a GVariant instance with a string type. This includes the types G_VARIANT_TYPE_STRING, G_VARIANT_TYPE_OBJECT_PATH and G_VARIANT_TYPE_SIGNATURE.

If *length* is non-NULL then the length of the string (in bytes) is returned there. For trusted values, this information is already known. For untrusted values, a strlen() will be performed.

It is an error to call this function with a *value* of any type other than those three.

The return value remains valid as long as *value* exists.

value :
a string GVariant instance

length :
a pointer to a gsize, to store the length

Returns :
the constant string

g_variant_dup_string ()

```
gchar *
g_variant_dup_string (GVariant *value,
                      gsize *length);
```

Similar to g_variant_get_string() except that instead of returning a constant string, the string is duplicated.

The return value must be freed using g_free().

value :
a string GVariant instance

length :
a pointer to a gsize, to store the length

Returns :
a newly allocated string

g_variant_get_variant ()

```
GVariant *
g_variant_get_variant (GVariant *value);
```

Unboxes *value*. The result is the GVariant instance that was contained in *value*.

value :
a variant GVariance instance

Returns **:**
the item contained in the variant

g_variant_n_children ()

```
gsize
g_variant_n_children (GVariant *value);
```

Determines the number of children in a container GVariant instance. This includes variants, maybes, arrays, structures and dictionary entries. It is an error to call this function on any other type of GVariant.

For variants, the return value is always 1. For maybes, it is always zero or one. For arrays, it is the length of the array. For structures it is the number of structure items (which depends only on the type). For dictionary entries, it is always 2.

This function never fails. TS

value **:**
a container GVariant

Returns **:**
the number of children in the container

g_variant_get_child ()

```
GVariant *
g_variant_get_child (GVariant *value,
                     gsize index);
```

Reads a child item out of a container GVariant instance. This includes variants, maybes, arrays, structures and dictionary entries. It is an error to call this function on any other type of GVariant.

It is an error if *index* is greater than the number of child items in the container. See g_variant_n_children().

This function never fails.

value :
a container GVariant

index :
the index of the child to fetch

Returns :
the child at the specified index

g_variant_get_fixed ()

```
gconstpointer
g_variant_get_fixed (GVariant *value,
                     gsize size);
```

Gets a pointer to the data of a fixed sized GVariant instance. This pointer can be treated as a pointer to the equivalent C stucture type and accessed directly. The data is in machine byte order.

size must be equal to the fixed size of the type of *value*. It isn't used for anything, but serves as a sanity check to ensure the user of this function will be able to make sense of the data they receive a pointer to.

This function may return NULL if *size* is zero.

value :
a GVariant

size :
the size of *value*

Returns :
a pointer to the fixed-sized data

g_variant_get_fixed_array ()

```
gconstpointer
g_variant_get_fixed_array (GVariant *value,
                           gsize elem_size,
                           gsize *length);
```

Gets a pointer to the data of an array of fixed sized GVariant instances. This pointer can be treated as a pointer to an array of the equivalent C structure type and accessed directly. The data is in machine byte order.

elem_size must be equal to the fixed size of the element type of *value*. It isn't used for anything, but serves as a sanity check to ensure the user of this function will be able to make sense of the data they receive a pointer to.

length is set equal to the number of items in the array, so that the size of the memory region returned is *elem_size* times *length*.

This function may return NULL if either *elem_size* or *length* is zero.

value **:**
an array GVariant

elem_size **:**
the size of one array element

length **:**
a pointer to the length of the array, or NULL

Returns **:**
a pointer to the array data

GVariantIter

```
typedef struct OPAQUE_TYPE__GVariantIter GVariantIter;
```

An opaque structure type used to iterate over a container GVariant instance.

The iter must be initialised with a call to g_variant_iter_init() before using it. After that, g_variant_iter_next() will return the child values, in order.

The iter may maintain a reference to the container GVariant until g_variant_iter_next() returns NULL. For this reason, it is essential that you call g_variant_iter_next() until NULL is returned. If you want to abort iterating part way through then use g_variant_iter_cancel().

g_variant_iter_init ()

```
gsize
g_variant_iter_init (GVariantIter *iter,
                     GVariant *value);
```

Initialises the fields of a GVariantIter and perpare to iterate over the contents of *value*.

iter is allowed to be completely uninitialised prior to this call; it does not, for example, have to be cleared to zeros. For this reason, if *iter* was already being used, you should first cancel it with g_variant_iter_cancel().

After this call, *iter* holds a reference to *value*. The reference will be automatically dropped once all values have been iterated over or manually by calling g_variant_iter_cancel().

This function returns the number of times that g_variant_iter_next() will return non-NULL. You're not expected to use this value, but it's there incase you wanted to know.

iter **:**
a GVariantIter

value **:**
a container GVariant instance

Returns **:**
the number of items in the container

g_variant_iter_next ()

```
GVariant *
g_variant_iter_next (GVariantIter *iter);
```

Retreives the next child value from *iter*. In the event that no more child values exist, NULL is returned and *iter* drops its reference to the value that it was created with.

The return value of this function is internally cached by the *iter*, so you don't have to unref it when you're done. For this reason, thought, it is important to ensure that you call `g_variant_iter_next()` one last time, even if you know the number of items in the container.

It is permissable to call this function on a cancelled iter, in which case NULL is returned and nothing else happens.

iter **:**
a `GVariantIter`

Returns **:**
a `GVariant` for the next child

g_variant_iter_cancel ()

```
void
g_variant_iter_cancel (GVariantIter *iter);
```

Causes *iter* to drop its reference to the container that it was created with. You need to call this on an iter if, for some reason, you stopped iterating before reading the end.

You do not need to call this in the normal case of visiting all of the elements. In this case, the reference will be automatically dropped by `g_variant_iter_next()` just before it returns NULL.

It is permissable to call this function more than once on the same iter. It is permissable to call this function after the last value.

***iter*:**
a GVariantIter

g_variant_iter_was_cancelled ()

```
gboolean
g_variant_iter_was_cancelled (GVariantIter *iter);
```

Determines if g_variant_iter_cancel() was called on *iter*.

***iter*:**
a GVariantIter

***Returns*:**
TRUE if g_variant_iter_cancel() was called

g_variant_iterate ()

```
gboolean
g_variant_iterate (GVariantIter *iter,
                   const gchar *format_string,
                   ...);
```

Retreives the next child value from *iter* and deconstructs it according to *format_string*. This call is sort of like calling g_variant_iter_next() and g_variant_get().

This function does something else, though: on all but the first call (including on the last call, which returns FALSE) the values allocated by the previous call will be freed. This allows you to iterate without ever freeing anything yourself. In the case of GVariant * arguments, they are unref'd and in the case of GVariantIter arguments, they are cancelled.

Note that strings are not freed since (as with g_variant_get()) they are constant pointers to internal GVariant data.

This function might be used as follows:

```
{
  const gchar *key, *value;
  GVariantIter iter;
  ...

  while (g_variant_iterate (iter, "{ss}", &key, &value))
    printf ("dict['%s'] = '%s'\n", key, value);
}
```

iter :
a GVariantIter

format_string :
a format string

... :
arguments, as per *format_string*

Returns :
TRUE if a child was fetched or FALSE if not

GVariantBuilder

```
typedef struct OPAQUE_TYPE__GVariantBuilder GVariantBuilder;
```

An opaque type used to build container GVariant instances one child value at a time.

G_VARIANT_BUILDER_ERROR

```
#define G_VARIANT_BUILDER_ERROR
```

Error domain for GVariantBuilder. Errors in this domain will be from the GVariantBuilderError enumeration. See GError for information on error domains.

enum GVariantBuilderError

```
typedef enum
{
  G_VARIANT_BUILDER_ERROR_TOO_MANY,
  G_VARIANT_BUILDER_ERROR_TOO_FEW,
  G_VARIANT_BUILDER_ERROR_INFER,
  G_VARIANT_BUILDER_ERROR_TYPE
} GVariantBuilderError;
```

Errors codes returned by g_variant_builder_check_add() and g_variant_builder_check_end().

G_VARIANT_BUILDER_ERROR_TOO_MANY
too many items have been added (returned by g_variant_builder_check_add())

G_VARIANT_BUILDER_ERROR_TOO_FEW
too few items have been added (returned by g_variant_builder_check_end())

G_VARIANT_BUILDER_ERROR_INFER
unable to infer the type of an array or maybe (returned by g_variant_builder_check_end())

G_VARIANT_BUILDER_ERROR_TYPE
the value is of the incorrect type (returned by g_variant_builder_check_add())

g_variant_builder_cancel ()

```
void
g_variant_builder_cancel (GVariantBuilder *builder);
```

Cancels the build process. All memory associated with *builder* is freed. If the builder was created with g_variant_builder_open() then all ancestors are also freed.

builder :
a GVariantBuilder

g_variant_builder_add ()

```
void
g_variant_builder_add (GVariantBuilder *builder,
                       const gchar *format_string,
                       ...);
```

Adds to a GVariantBuilder.

This call is a convenience wrapper that is exactly equivalent to calling g_variant_new() followed by g_variant_builder_add_value().

This function might be used as follows:

```
GVariant *
make_pointless_dictionary (void)
{
  GVariantBuilder *builder;
  int i;

  builder = g_variant_builder_new (G_VARIANT_TYPE_CLASS_ARRAY,
                                   NULL);
  for (i = 0; i < 16; i++)
    {
      char buf[3];

      sprintf (buf, "%d", i);
      g_variant_builder_add (builder, "{is}", i, buf);
    }

  return g_variant_builder_end (builder);
}
```

builder :
a GVariantBuilder

format_string :
a GVariant varargs format string

... :
arguments, as per *format_string*

g_variant_builder_add_value ()

```
void
g_variant_builder_add_value (GVariantBuilder *builder,
                             GVariant *value);
```

Adds *value* to *builder*.

It is an error to call this function if *builder* has an outstanding child. It is an error to call this function in any case that g_variant_builder_check_add() would return FALSE.

builder :
a GVariantBuilder

value :
a GVariant

g_variant_builder_check_add ()

```
gboolean
g_variant_builder_check_add (GVariantBuilder *builder,
                             GVariantTypeClass class,
                             const GVariantType *type,
                             GError **error);
```

Does all sorts of checks to ensure that it is safe to call g_variant_builder_add() or g_variant_builder_open().

It is an error to call this function if *builder* has a child (ie: g_variant_builder_open() has been used on *builder* and g_variant_builder_close() has not yet been called).

It is an error to call this function with an invalid *class* (including G_VARIANT_TYPE_CLASS_INVALID) or a class that's not the smallest class for some concrete type (for example, G_VARIANT_TYPE_CLASS_ALL).

If *type* is non-NULL this function first checks that it is a member of *class* (except, as with g_variant_builder_new(), if *class* is G_VARIANT_TYPE_CLASS_VARIANT then any *type* is OK).

The function then checks if any child of class *class* (and type *type*, if given) would be suitable for adding to the builder. If *type* is non-NULL and is non-concrete then all concrete types matching *type* must be suitable for adding (ie: *type* must be equal to or less general than the type expected by the builder).

In the case of an array that already has at least one item in it, this function performs an additional check to ensure that *class* and *type* match the items already in the array. *type*, if given, need not be concrete in order for this check to pass.

Errors are flagged in the event that the builder contains too many items or the addition would cause a type error.

If *class* is specified and is a container type and *type* is not given then there is no guarantee that adding a value of that class would not fail. Calling g_variant_builder_open() with that *class* (and *type* as NULL) would succeed, though.

In the case that any error is detected *error* is set and FALSE is returned.

builder **:**
a GVariantBuilder

class **:**
a GVariantTypeClass

type :
a GVariantType, or NULL

error :
a GError

Returns :
TRUE if adding is safe

g_variant_builder_check_end ()

```
gboolean
g_variant_builder_check_end (GVariantBuilder *builder,
                             GError **error);
```

Checks if a call to g_variant_builder_end() or g_variant_builder_close() would succeed.

It is an error to call this function if *builder* has a child (ie: g_variant_builder_open() has been used on *builder* and g_variant_builder_close() has not yet been called).

This function checks that a sufficient number of items have been added to the builder. For dictionary entries, for example, it ensures that 2 items were added.

This function also checks that array and maybe builders that were created without concrete type information contain at least one item (without which it would be impossible to infer the concrete type).

If some sort of error (either too few items were added or type inference is not possible) prevents the builder from being ended then FALSE is returned and *error* is set.

builder :
a GVariantBuilder

error :
a GError

Returns **:**
TRUE if ending is safe

g_variant_builder_close ()

```
GVariantBuilder *
g_variant_builder_close (GVariantBuilder *child);
```

This function closes a builder that was created with a call to `g_variant_builder_open()`.

It is an error to call this function on a builder that was created using `g_variant_builder_new()`. It is an error to call this function if *child* has an outstanding child. It is an error to call this function in any case that `g_variant_builder_check_end()` would return FALSE.

child **:**
a `GVariantBuilder`

Returns **:**
the original parent of *child*

g_variant_builder_end ()

```
GVariant *
g_variant_builder_end (GVariantBuilder *builder);
```

Ends the builder process and returns the constructed value.

It is an error to call this function on a `GVariantBuilder` created by a call to `g_variant_builder_open()`. It is an error to call this function if *builder* has an outstanding child. It is an error to call this function in any case that `g_variant_builder_check_end()` would return FALSE.

builder **:**
a `GVariantBuilder`

Returns **:**
a new, floating, GVariant

g_variant_builder_new ()

```
GVariantBuilder *
g_variant_builder_new (GVariantTypeClass class,
                       const GVariantType *type);
```

Creates a new GVariantBuilder.

class must be specified and must be a container type.

If *type* is given, it constrains the child values that it is permissible to add. If *class* is not G_VARIANT_TYPE_CLASS_VARIANT then *type* must be contained in *class* and will match the type of the final value. If *class* is G_VARIANT_TYPE_CLASS_VARIANT then *type* must match the value that must be added to the variant.

After the builder is created, values are added using g_variant_builder_add_value().

After all the child values are added, g_variant_builder_end() ends the process.

class **:**
a container GVariantTypeClass

type **:**
a type contained in *class*, or NULL

Returns **:**
a GVariantBuilder

g_variant_builder_open ()

```
GVariantBuilder *
g_variant_builder_open (GVariantBuilder *parent,
                        GVariantTypeClass class,
                        const GVariantType *type);
```

Opens a subcontainer inside the given *parent*.

It is not permissible to use any other builder calls with *parent* until @g_variant_builder_close() is called on the return value of this function.

It is an error to call this function if *parent* has an outstanding child. It is an error to call this function in any case that g_variant_builder_check_add() would return FALSE. It is an error to call this function in any case that it's an error to call g_variant_builder_new().

If *type* is NULL and *parent* was given type information, that information is passed down to the subcontainer and constrains what values may be added to it.

parent **:**
a GVariantBuilder

class **:**
a GVariantTypeClass

type **:**
a GVariantType, or NULL

Returns **:**
a new (child) GVariantBuilder

g_variant_markup_print ()

```
GString *
g_variant_markup_print (GVariant *value,
                        GString *string,
                        gboolean newlines,
                        gint indentation,
                        gint tabstop);
```

Pretty-prints *value* as an XML document fragment.

If *string* is non-NULL then it is appended to and returned. Else, a new empty GString is allocated and it is returned.

The *newlines*, *indentation* and *tabstop* parameters control the whitespace that is emitted as part of the document.

If *newlines* is TRUE, then newline characters will be printed where appropriate.

If *indentation* is non-zero then this is the number of spaces that are printed before the first and last tag. If *tabstop* is non-zero then this is the number of additional spaces that are added for each level of nesting.

value **:**
a GVariant

string **:**
a GString, or NULL

newlines **:**
TRUE if newlines should be printed

indentation **:**
the current indentation level

tabstop **:**
the number of spaces per indentation level

Returns **:**
a GString containing the XML fragment

g_variant_markup_parse ()

```
GVariant *
g_variant_markup_parse (const gchar *text,
                        gssize text_len,
                        const GVariantType *type,
                        GError **error);
```

One of the three interfaces to the GVariant markup parser. For information about the others, see g_variant_markup_subparser_start() and g_variant_markup_parse_context_new().

You should use this interface if you have an XML document representing a GVariant value entirely contained within a single string.

text should be the full text of the document. If *text_len* is not -1 then it gives the length of *text* (similar to g_markup_parse_context_parse()).

If *type* is non-NULL then it constrains the permissible types that the root element may have. It also serves to hint the parser about the type of this element (and may, for example, resolve errors caused by the inability to infer the type).

In the case of an error then NULL is returned and *error* is set to a description of the error condition. This function is robust against arbitrary input; all error conditions are reported via *error* -- your program will never abort.

text **:**
the self-contained document to parse

text_len **:**
the length of *text*, or -1

type **:**
a GVariantType constraining the type of the root element

error **:**
a GError

Returns **:**
a new GVariant, or NULL in case of an error

g_variant_markup_subparser_start ()

```
void
g_variant_markup_subparser_start (GMarkupParseContext *context,
                                  const GVariantType *type);
```

One of the three interfaces to the GVariant markup parser. For information about the others, see g_variant_markup_parse() and g_variant_markup_parse_context_new().

You should use this interface if you are parsing an XML document using GMarkupParser and that document contains an embedded GVariant among the markup.

You should call this function from the start_element handler of your parser for the element containing the markup for the GVariant and then return immediately. The next call to your parser will either be an error condition or a call to the end_element handler for the tag matching the start tag. From here, you should call g_variant_markup_parser_pop to collect the result.

For example, if your document contained sections like this:

```
<my-value>
  <int32>42</int32>
</my-value>
```

Then your handlers might contain code like:

```
start_element()
{
  if (strcmp (element_name, "my-value") == 0)
    g_variant_markup_subparser_start (context, NULL);
  else
    {
      ...
    }

}
```

```
end_element()
{
  if (strcmp (element_name, "my-value") == 0)
    {
      GVariant *value;

      if (!(value = g_variant_markup_subparser_pop (context, error)))
        return;

      ...
    }
  else
    {
      ...
    }
}
```

If *type* is non-NULL then it constrains the permissible types that the root element may have. It also serves to hint the parser about the type of this element (and may, for example, resolve errors caused by the inability to infer the type).

This call never fails, but it is possible that the call to `g_variant_markup_subparser_end()` will.

context **:**
a `GMarkupParseContext`

type **:**
a `GVariantType` constraining the type of the root element

g_variant_markup_subparser_end ()

```
GVariant *
g_variant_markup_subparser_end (GMarkupParseContext *context,
                                GError **error);
```

Ends the subparser started by `g_variant_markup_subparser_start()` and collects the results.

You must call this function from the end_element handler invocation corresponding to the start_element handler invocation from which g_variant_markup_subparser_start() was called. This will be the first end_handler invocation that is received after calling g_variant_markup_subparser_start().

If an error occured while processing tags in the subparser then your end_element handler will not be invoked at all and you should not call this function.

The only time this function will fail is if no value was contained between the start and ending tags.

context **:**
a GMarkupParseContext

error **:**
the end_element handler *error*, passed through

Returns **:**
a GVariant or NULL.

g_variant_markup_parse_context_new ()

```
GMarkupParseContext *
g_variant_markup_parse_context_new (GMarkupParseFlags flags,
                                    const GVariantType *type);
```

One of the three interfaces to the GVariant markup parser. For information about the others, see g_variant_markup_parse() and g_variant_markup_subparser_start().

You should use this interface if you have an XML document that you want to feed to the parser in chunks.

This call creates a GMarkupParseContext setup for parsing a GVariant XML document. You feed the document to the parser one chunk at a time using the normal g_markup_parse_context_parse() call. After the entire document

is fed, you call g_variant_markup_parse_context_end() to free the context and retreive the value.

If *type* is non-NULL then it constrains the permissible types that the root element may have. It also serves to hint the parser about the type of this element (and may, for example, resolve errors caused by the inability to infer the type).

If you want to abort parsing, you should free the context using g_markup_parse_context_free().

flags **:**
GMarkupParseFlags

type **:**
a GVariantType constraining the type of the root element

Returns **:**
a new GMarkupParseContext

g_variant_markup_parse_context_end ()

```
GVariant *
g_variant_markup_parse_context_end (GMarkupParseContext *context,
                                    GError **error);
```

Ends the parsing started with g_variant_markup_parse_context_new().

context must have been the result of a previous call to g_variant_markup_parse_context_new().

This function calls g_markup_parse_context_end_parse() and g_markup_parse_context_free() for you.

If the parsing was successful, a GVariant is returned. Otherwise, NULL is returned and *error* is set accordingly.

context **:**
a GMarkupParseContext

error **:**
a GError

Returns **:**
a GVariant, or NULL

GVariant-loadstore

Synopsis

```
enum                GVariantFlags;
void                g_variant_store                 (GVariant *value,
                                                     gpointer data);
gconstpointer       g_variant_get_data              (GVariant *value);
gsize               g_variant_get_size              (GVariant *value);
GVariant*           g_variant_load                  (const GVariantType *type,
                                                     gconstpointer data,
                                                     gsize size,
                                                     GVariantFlags flags);
GVariant*           g_variant_from_slice            (const GVariantType *type,
                                                     gpointer slice,
                                                     gsize size,
                                                     GVariantFlags flags);
GVariant*           g_variant_from_data             (const GVariantType *type,
                                                     gconstpointer data,
                                                     gsize size,
                                                     GVariantFlags flags,
                                                     GDestroyNotify notify,
                                                     gpointer user_data);
```

Description

Details

enum GVariantFlags

```
typedef enum
{
  G_VARIANT_TRUSTED             = 0x00010000,
  G_VARIANT_LAZY_BYTESWAP       = 0x00020000,
} GVariantFlags;
```

g_variant_store ()

```
void
g_variant_store (GVariant *value,
                 gpointer data);
```

Stores the serialised form of *variant* at *data*. *data* should be serialised enough. See `g_variant_get_size()`.

The stored data is in machine native byte order but may not be in fully-normalised form if read from an untrusted source.

This function is approximately O(n) in the size of *data*.

This function never fails.

value **:**
the GVariant to store

data **:**
the location to store the serialised data at

g_variant_get_data ()

```
gconstpointer
g_variant_get_data (GVariant *value);
```

Returns a pointer to the serialised form of a GVariant instance. The returned data is in machine native byte order but may not be in fully-normalised form if read from an untrusted source. The returned data must not be freed; it remains valid for as long as *value* exists.

In the case that *value* is already in serialised form, this function is O(1). If the value is not already in serialised form, serialisation occurs implicitly and is approximately O(n) in the size of the result.

This function never fails.

value **:**
a GVariant instance

Returns **:**
the serialised form of *value*

g_variant_get_size ()

```
gsize
g_variant_get_size (GVariant *value);
```

Determines the number of bytes that would be required to store *value* with `g_variant_store()`.

In the case that *value* is already in serialised form or the size has already been calculated (ie: this function has been called before) then this function is O(1). Otherwise, the size is calculated, an operation which is approximately O(n) in the number of values involved.

This function never fails.

value **:**
a `GVariant` instance

Returns **:**
the serialised size of *value*

g_variant_load ()

```
GVariant *
g_variant_load (const GVariantType *type,
                gconstpointer data,
                gsize size,
                GVariantFlags flags);
```

Creates a new `GVariant` instance. *data* is copied. For a more efficient way to create `GVariant` instances, see `g_variant_from_slice()` or `g_variant_from_data()`.

This function is O(n) in the size of *data*.

This function never fails.

type **:**
the `GVariantType` of the new variant

data **:**
the serialised `GVariant` data to load

size :
the size of *data*

flags :
zero or more GVariantFlags

Returns :
a new GVariant instance

g_variant_from_slice ()

```
GVariant *
g_variant_from_slice (const GVariantType *type,
                      gpointer slice,
                      gsize size,
                      GVariantFlags flags);
```

Creates a GVariant instance from a memory slice. Ownership of the memory slice is assumed. This function allows efficiently creating GVariant instances with data that is, for example, read over a socket.

If *type* is NULL then *data* is assumed to have the type G_VARIANT_TYPE_VARIANT and the return value is the value extracted from that variant.

This function never fails.

type :
the GVariantType of the new variant

slice :
a pointer to a GSlice-allocated region

size :
the size of *slice*

flags :
zero or more GVariantFlags

Returns :
a new GVariant instance

g_variant_from_data ()

```
GVariant *
g_variant_from_data (const GVariantType *type,
                     gconstpointer data,
                     gsize size,
                     GVariantFlags flags,
                     GDestroyNotify notify,
                     gpointer user_data);
```

Creates a GVariant instance from serialised data. The data is not copied. When the data is no longer required (which may be before or after the return value is freed) *notify* is called. *notify* may even be called before this function returns.

If *type* is NULL then *data* is assumed to have the type G_VARIANT_TYPE_VARIANT and the return value is the value extracted from that variant.

This function never fails.

type :
the GVariantType of the new variant

data :
a pointer to the serialised data

size :
the size of *data*

flags :
zero or more GVariantFlags

notify :
a function to call when *data* is no longer needed

user_data **:**
a `gpointer` to pass to *notify*

Returns **:**
a new `GVariant` instance

Appendix B

Synchronisation Primitives

A GVariant instance is a very small structure. It uses only 24 bytes of memory on 32-bit systems.

On 32-bit systems, a GStaticMutex lock, as made available in GLib is 28 bytes. A GMutex is also 24 bytes (being implemented as a POSIX pthread_mutex_t which is the same size) and it also requires allocation of a separate memory region (increasing memory management overhead) and would also necessitate adding another pointer to the GVariant structure (which would increase the size of a GVariant instance to 32 bytes)[1].

Associating one of these existing primitives with each GVariant instance would more than double the amount of memory used as overhead by GVariant. There are several places in GVariant, however, where access to a given instance must be limited to a single thread. For this reason, alternative solutions were sought out.

One potential solution that was considered for this problem was to use code locks. A single lock would be allocated and used to ensure

[1] Memory in GLib is allocated via the slice allocator, which allocates memory regions with a size granularity of 2 pointer sizes. On 32-bit machines this is 8 bytes.

that sensitive GVariant code is executing in no more than one thread. This solution would result in an excessive amount of lock contention, however, since even if threads were accessing totally separate instances they will still block on each other.

Some form of finer-grained locking is required. A number of alternative solutions were evaluated and finally one was settled one: a 1-bit mutex lock.

The 1-bit mutex has been proposed for[2] (and is likely to be included in) the next version of GLib as a general purpose interface. It will be available as the functions `g_bit_lock()` and `g_bit_unlock()`.

Implementation of this lock is described here.

B.1 Atomic integer operations

GLib contains a small library of atomic integer access functions. These functions can be used to perform a small range of memory operations (such as adding to an integer) atomically. Since there is no portable way to perform atomic integer access in C, these functions are implemented in assembly language for each of the machines on which GLib runs.

The implementation of the 1-bit mutex relies on two of these operations.

`g_atomic_int_get()`
: reads the value of the integer. This function also acts as a memory barrier on platforms that require it for cross-processor consistency.

`g_atomic_int_compare_and_exchange()`
: first ensures that the value of the integer has a certain expected value, then sets it equal to a new value. A boolean is returned to indicate if the operation succeeded. This function also acts as a memory barrier.

2

B.2 Futex

In order to implement the 1-bit mutex, a lower level synchronisation primitive was required. This primitive is the `futex(2)` system call provided by the Linux kernel.

The name "futex" originated from the phrase "fast userspace mutex". This name is misleading, however, since a futex is not a mutex at all — it is merely a useful tool for implementing one.

A futex is actually a sort of wait queue. Threads register an interest in receiving wake-up signals at a given virtual memory address. The thread blocks until another thread sends a wake-up message for the same address (at which point only one waiting thread wakes up). Before the thread blocks, the given memory address is atomically checked to ensure that it contains a value specified by the user.

The `futex(2)` system call is only available on Linux systems and isn't portable at all. Linux is the primary operating system on which GNOME is used, however, so most users will have this efficient native implementation available to them.

B.2.1 Emulating futex

On Linux, we are able to use the `futex(2)` system call to get the desired functionality. GLib is intended to be portable to a wide range of systems, however. On other systems, we must use other existing synchronisation primitives to implement futex-like functionality for ourselves.

GLib contains the GCond data type — a message queue that supports blocking and wake-up notification much like a futex. GCond, however, must be allocated before it can be used.

The futex emulation code simply maintains a linked list mapping addresses that are being waited on to GCond queues. The futex wait and wake operations on a given address and then implemented using the GCond wait and wake operations on the appropriate condition queue.

All of the operations on the linked list mapping are performed while holding a traditional mutex lock.

This may sound very complicated and time consuming, but it is important to understand that futex calls are actually very rare — they are only used to resolve the contended case of lock acquisition and release.

B.3 The 1-bit mutex

The 1-bit mutex implements a mutex lock using only a single bit in an integer value. This bit is (easily) available in the state register of a GVariant instance, allowing for per-instance locking.

The process of acquiring the lock uses the GLib atomic integer operations. First, the value of the entire integer is atomically read. If the bit used to represent the lock is unset then the old value is compare-and-exchanged with a new value which has the bit set. If another thread were attempting to acquire the lock at the same time, this operation would fail (since the memory address would not compare equal to the old value). If this operation succeeds then the lock has been acquired and the function returns.

If the lock bit was set then we are dealing with the contended case. We use a futex wait operation to sleep on the address (while checking that the value at the address is still what we expect — and therefore has not been unlocked in the meantime).

To unlock, the same read/compare-and-exchange sequence is used to unset the lock bit. A futex wake operation is invoked on the address of the integer to wake any threads that might be waiting to acquire the lock.

B.3.1 An optimisation: contention counters

The 1-bit mutex lock implementation described above suffers from a small wart: the futex wake operation is invoked even in the case where nobody is waiting. This call is harmless, but takes time to execute. On Linux this implies a (fast) system call on each unlock. On other systems it implies a search through a (probably empty) linked list.

A simple optimisation has been implemented to avoid this extra call in the vast majority of cases.

The 1-bit mutex keeps an internal static array of "contention counters". The length of this array is a prime number (currently 11). The address of the integer being used for the lock is divided by this prime number and the remainder is taken as an index into the array. This provides a constant factor reduction in the number of instances that are sharing a single contention counter.

Before waiting on a futex the contention counter associated with the wait address is incremented. After returning from a wait, the contention counter is decremented. Before the unlock code executes a wake operation, the contention counter for the address is checked.

Excepting the case of sharing a contention counter with another contended instance, this means that futex wake calls are only ever executed when another thread is actually waiting for the lock. This exceptional case is expected to be hilariously rare and even if it occurs, the extra futex call is harmless.

B.4 Future work: better bit operations

The use of the read/compare-and-exchange sequences in the described implementation are necessitated by the lack of atomic bit test-and-set or test-and-clear functions in GLib. On Intel systems (on which GNOME runs most commonly) these operations can be implemented with single hardware instructions.

At this time, these operations haven't been implemented. To do so properly would require knowledge of the assembly languages for every machine on which GLib runs — 32 and 64-bit version of x86, ARM, sparc, PowerPC and S390.

B.5 Other approaches

Some of the other approaches that were considered are detailed here.

B.5.1 "Friendly" spinning

A very simple variation on using a single bit for locking is to attempt to acquire the lock and yield to the scheduler if that fails. This is done in a loop until the lock is acquired.

The idea is that yielding to the scheduler will allow the thread holding the lock to run to completion. This is only slightly better, however, than an unmodified spinlock and is definitely to be considered "evil".

B.5.2 An array of mutexes

This alternate approach takes a cue from how a number of different condition counters are used to lower the chance of contended instances causing emission of extra futex wake calls in the 1-bit mutex.

Instead of a mutex being associated with each instance, an internal static array of mutexes would be used. The length of the array would be a prime number, and the address of the GVariant instance would decide which lock is acquired to protect that instance. This would provide a constant factor reduction in contention while still ensuring that only one thread could access a particular instance (at the cost of preventing access to some others).

In some cases, however (for example, when serialising an entire tree) several GVariant instances need to be locked at the same time (with

the same thread holding all locks). If two of these instances aliased in the mutex array, the program would deadlock. Contrast this with the contention counter array in the 1-bit mutex implementation where a conflict merely results in a small amount of unnecessary work being done.

This could be resolved by using recursive mutexes. Unfortunately, this approach conceals an even more insidious problem. If in one thread an operation involved locking two instances in order and in another thread another operation involved locking two other instances in order, and the instances aliased in the mutex array such that the locks were acquired in reverse order with respect to each other, the program would deadlock due to a lock order inversion.

Without making any statement about whether a workaround could be developed to deal with this situation, the entire approach has ultimately been abandoned as fundamentally inelegant.

B.5.3 The 2-bit mutex

Instead of using only a single bit plus contention counters a more direct approach could have been taken: use a second bit to indicate the contended case.

The second thread to attempt to acquire the lock would set the "contended" bit before going to sleep waiting for notification. Then notification would only be sent if the contended bit was set.

One problem with this approach is in knowing how many threads are currently blocked. The contended bit can never be safely unset without knowing this. The futex wake call returns the number of processes that were woken but in order to discover that zero processes were sleeping, one extra unnecessary futex call needs to be issued. This would definitely cause extra calls compared to the contention counter approach since an extra call would occur in every case of contention.

Also, since the 1-bit mutex has been included as an external API in GLib, having it use only 1-bit makes it more appealing and easier to understand to other library users.

Finally, when holding the 1-bit mutex, it is safe to perform non-atomic bit operations on the same integer that contains the bit (for example, to set other flags). This is because, if the lock bit is held by the executing thread, it can be sure that no other bits will be modified until the lock is released. Bringing a contention bit into the situation complicates things — the integer can be modified by another thread at any time by the adding of the contention bit. Atomic integer operations would have to be used throughout.

Appendix C

Conditions

This appendix offers details on the implementation of the condition machinery that lies at the core of the implementation of GVariant. A list of all of the conditions is also provided.

For a description of conditions, see Section 10.3.1.

C.1 List of conditions

The following list represents all of the conditions that are currently defined in the implementation of GVariant.

For each condition, the enabling precondition predicate is given (as described in Section 10.3.1). Each condition also has a list of other conditions that it implies, that it forbids, and that its absence implies. This list is used only for runtime assertion checking. Due to logical equivalences, forbids and absence-implies are symmetric: if one condition forbids a condition or implies it with its absence then the reverse must be true.

C.1.1 CONDITION_SERIALISED (*ser*)

- enabling precondition: *sk*
- implies: *sk*
- forbids: *not*
- absence implies: *nat* ∧ *ind*

There are two main types of instances: serialised form and tree form.

Instances that are serialised have all of the data associated with their value encoded as a single array of bytes at one location in memory. Non-container values are always serialised.

Container values, if not serialised, are stored in tree form; the instance is an array of values pointing to the child values of the container.

The enabling function involves creating a memory region of the appropriate size and serialising the children into that memory region. The memory region is then set as the serialised data of the instance.

sk is an enabling precondition because the amount of memory to allocate for the new buffer must be known.

nat ∧ *ind* are implied by the absence of *ser* since an instance in tree form will always be in native byte order once serialised and doesn't depend on serialised data from any "source instance".

C.1.2 CONDITION_SIZE_KNOWN (*sk*)

- enabling precondition: ⊤
- implies: ⊤
- forbids: ¬

- absence implies: $\top$

The size of an instance is known if the size of the data, if it were to be serialised, is known. Implicitly, serialised instances are always size-known (since the serialised data is at hand), so this condition is only interesting for instances in tree form.

The enabling function involves invoking the serialiser on the child values in the tree to predetermine the amount of memory that would be required to serialise them.

Note that it is possible, even if the size is known, that the size may change. `CONDITION_SIZE_VALID` is the condition that the size will never change.

C.1.3 `CONDITION_INDEPENDENT` (*ind*)

- enabling precondition: $\neg nat$
- implies: $\top$
- forbids: $\bot$
- absence implies: *ser*

An instance is independent if any serialised data that the instance is using belongs to that instance. An instance that is not independent is using data that belongs to another (parent) instance or to the user.

The enabling function involves allocating a new memory region, making a copy of the buffer (from the source), and using that new buffer. If it was discovered that the source data was byteswapped during the copy then the function fails.

$\neg nat$ as an enabling precondition has a number of effects. First, it implies that the instance is serialised and has its size known (since these states are implied by $\neg nat$).

Second, *nat* is a precondition to exposing the serialised data to the user. If the user has seen the data then we cannot run the enabling function since the address at which the data resides will change (thus invalidating the pointer that was given to the user). The only way to be sure that the user has not seen the data is if *nat* is not set.

Absence of *ind* implies *ser* since if serialised data from another instance is being used then the instance is obviously in serialised form.

C.1.4 CONDITION_FIXED_SIZE (*fix*)

- enabling precondition: ⊤
- implies: ⊤
- forbids: ¬
- absence implies: ⊤

An instance is known fixed-sized if the type of the instance is recognised as being a type where all values have the same size (for example: floating point values, but not strings).

The enabling function involves checking the type of the instance. The transition function will fail if the type is not a fixed-size type. For simplicity of implementation, this condition has been kept separate from `CONDITION_SIZE_KNOWN`. No size information is actually collected or stored by this enabling function.

Fixed sized values are interesting because, as mentioned in Section 7.1, they can always be safely byteswapped, even when not in normal form.

C.1.5 CONDITION_SIZE_VALID (*sv*)

- enabling precondition: (*sk* ∧ *fix*) ∨ (*sk* ∧ *tru*) ∨ (*sk* ∧ *nat*)
- implies: *sk*

- forbids: $\bot$
- absence implies: $\top$

The size of an instance can be known to be valid if the size is known and it is certain that the size will never change. The only thing that might potentially change the size is reconstruction, which doesn't happen to instances that are native trusted or fixed-sized. In any of these cases, the size of the instance will never change in the future.

This condition must be enabled before the serialised size of the instance is reported to the user since once the size is reported it must not change.

There is no enabling function; the enabling precondition is sufficient. Each of the clauses in the precondition represents a different path to being certain that the size of the serialised data of the instance will not change The clauses are ordered from least to most expensive to satisfy.

C.1.6 CONDITION_NATIVE (*nat*)

- enabling precondition: *sn* ∨ *bn* ∨ *rec*
- implies: $\top$
- forbids: *not*
- absence implies: *ser*

An instance is native if it is known to be in the byte order of the host machine.

This condition must be enabled before a pointer to the serialised data is given to the user since the user will expect the data to be in native byte order.

There is no enabling function; the enabling precondition is sufficient. Each of the clauses in the precondition represents a different path to

the serialised data being known to be in native byte order. The clauses are ordered from least to most expensive to satisfy.

C.1.7 CONDITION_SOURCE_NATIVE (*sn*)

- enabling precondition: $\neg ind \wedge \neg nat$
- implies: $nat \wedge ser$
- forbids: $ind \vee bn \vee rec \vee not$
- absence implies: $\top$

An instance is source-native if it is known that its source instance has been byteswapped to native byte order since the instance was created. If the data of the source (which is shared by this instance) is now in native byte order then this instance is as well.

The enabling function involves checking the *bn* condition on the source instance to see if it has been enabled. The function fails if the source instance does not have the *bn* condition enabled.

$\neg ind$ is a precondition since independent values have no source to check. $\neg nat$, because if the value is already in native byte order then it makes no sense to be performing this check.

C.1.8 CONDITION_BECAME_NATIVE (*bn*)

- enabling precondition: $(fix \wedge ind \wedge \neg nat) \vee (tru \wedge ind \wedge \neg\ nat)$
- implies: $nat \wedge ser \wedge ind$
- forbids: $sn \vee rec \vee not$
- absence implies: $\top$

An instance became native if it has converted itself to native byte order since it was created.

The enabling function involves byteswapping the serialised representation from non-native to native byte order.

ind is a precondition because we can only byteswap data that we "own". ¬*nat* because it makes no sense to byteswap data that is already in native byte order. *fix* ∨ *tru* is required because, as mentioned in Section 7.1, byteswapping serialised data is only safe if it is fixed-sized or in normal form.

C.1.9 CONDITION_TRUSTED (*tru*)

- enabling precondition: *st* ∨ *bt* ∨ *rec*
- implies: *tru* ∧ *ser*
- forbids: *bn* ∨ *rec* ∨ *not*
- absence implies: ⊤

An instance is trusted if its serialised data is known to be in normal form.

There is no enabling function; the enabling precondition is sufficient. Each of the clauses in the precondition represents a different path to the serialised data being known to be in normal form. The clauses are ordered from least to most expensive to satisfy.

C.1.10 CONDITION_SOURCE_TRUSTED (*st*)

- enabling precondition: ¬*tru* ∧ ¬*ind*
- implies: *tru* ∧ *ser*
- forbids: *ind* ∨ *bt* ∨ *rec* ∨ *not*

- absence implies: $\top$

An instance is source-trusted if it is known that its source instance has been scanned and found to be in normal form since the instance was created. If the data of the source (which is shared by this instance) is now trusted then the instance (which is contained in that data) must also be trusted.

The enabling function involves checking the *bt* condition on the source instance to see if it has been enabled. The function fails if the source instance does not have the *bt* condition enabled.

$\neg ind$ is a precondition since independent values have no source to check. $\neg tru$, because it makes no sense to perform this operation if the data is already trusted.

C.1.11 CONDITION_BECAME_TRUSTED (*bt*)

- enabling precondition: $ser \wedge \neg tru$
- implies: $tru \wedge ser$
- forbids: $bn \vee rec \vee not$
- absence implies: $\top$

An instance has become trusted if it has scanned itself and verified its data to be in normal form since it was created.

The enabling function involves scanning the serialised data of the instance to determine if it is in normal form. The function fails if an abnormality is found.

ser is a precondition since we must have serialised data to scan. $\neg tru$ is a precondition since it makes no sense to perform this operation if the serialised data is already trusted.

C.1.12 CONDITION_RECONSTRUCTED (*rec*)

- enabling precondition: *ser* ∧ *ind* ∧ ¬*nat* ∧ ¬*tru* ∧ ¬*fix*
- implies: *tru* ∧ *nat* ∧ *ind*
- forbids: *sn* ∨ *bn* ∨ *st* ∨ *bt* ∨ *not*
- absence implies: ⊤

Reconstruction is a gigantic hack to work around the fact that it's not possible to safely byteswap values that are not trusted and not fixed-size.

There is practically no reason for serialised data not to be in normal form unless "bad things" (eg: attacks on the system, etc.) are happening. For this reason, this case "should never happen" but is dealt with anyway to produce a friendlier API that is guaranteed never to fail.

Instead of byteswapping, the value is reconstructed using a slow deep copy method that essentially iterates and recurses over the structure of the value making a new, trusted, native copy of it.

Even though it is not possible that the data has been exposed to the user (since it's not in native endian) there exists the possibility that the instance is acting as source to other instances. For this reason, the original serialised data must be saved.

The instance is required to be independent because the pointer at which the old data is stored uses the same memory location usually used by the pointer to the source instance data in dependent instances. This implementation hack prevents increasing the GVariant structure size by 33% in order to deal with this "should never happen" case.

C.1.13 CONDITION_NOTIFY (*not*)

- enabling precondition: ⊥

- implies: ⊤
- forbids: everything else
- absence implies: ⊤

A notify instance does not represent a value. It is used, instead, as the source of a dependent instance that is using data provided by the user. When the notify instance is freed (indicating that the source data is no longer required) it dispatches a callback to the user to notify them that the data may be freed.

The notify condition is never enabled. It is set at creation time.

C.2 Condition machinery

The main entry point to the condition machinery is the call `g_variant_require_conditions()`. This function takes a set of conditions and instructs the machine to ensure that they are all satisfied.

In the most simple case, if the condition is already satisfied, the function immediately returns.

Next, if the condition's enabling prerequisite is currently satisfied, the enabling function of the condition (if any) is run, and the transition is enabled if it succeeds.

Failing that, the condition machinery searches for clauses in the enabling prerequisite with one false term. If it finds such a term then it invokes itself recursively, attempting to satisfy the missing prerequisites on the first such term that it finds.

If this fails, the condition machinery searches for clauses with two false terms, and so on. This heuristic typically results in a smaller amount of work being done. Also, since the disjunctive clauses in the enabling prerequisite are attempted in order the condition machine can additionally be tweaked to favour "less expensive" enabling functions.

If none of these attempts succeed, we restart the process because it will succeed a second time (see below).

C.2.1 Attempting to enable CONDITION_INDEPENDENT during a byteswap

The one case where the required condition may not be satisfied is when requiring CONDITION_NATIVE on a non-independent instance that was created with a non-native source. If the source instance is being byteswapped during the process of the condition machinery running then it can cause a failure. This will only occur if values are being simultaneously accessed from different threads.

When CONDITION_NATIVE is requested, it will attempt to satisfy the enabling precondition. The first clause includes CONDITION_SOURCE_NATIVE. This will fail if the source is not yet native. Meanwhile, the source byteswaps itself and is now in native byte order.

The next clause in the precondition for CONDITION_NATIVE includes CONDITION_BECAME_NATIVE which in turn requires CONDITION_INDEPENDENT. Since the source value is now in native byte order, however, the enabling function for CONDITION_INDEPENDENT will fail. This will cause the entire request to fail (since the only other clause also, transitively, requires independence).

Of course, retrying the transition will succeed since, this time, the CONDITION_SOURCE_NATIVE clause of the precondition will succeed (since conditions are never disabled).

C.3 Notes on thread safety

There are two fundamental problems with concurrent access to data structures:

- ensuring no two threads are making modifications at once

- ensuring no thread is making modifications during a read

The first issue is dealt with in a very simple way. The only functions that ever make modifications to a GVariant instance are the enabling functions for conditions. These functions are only ever invoked from the condition machine. The condition machine holds a per-instance mutex lock at all times that it is running. This effectively prevents any problems associated with concurrent modifications.

The second issue is somewhat more complicated. The simple way to solving this problem is by taking the mutex during all read accesses. This results in high lock contention. This problem can be partially alleviated through use of a reader-writer lock.

Still, with GVariant it's not possible to hold locks in all cases of access. Many of the API return pointers to the internal state of instances. This state will continue to be accessed after the calls return and locks can no longer be held.

The condition machine helps to solve these problems.

In the case that a function requires that a condition be disabled in order for an access to succeed, it uses the call `g_variant_forbid_condition()`. This call ensures that the condition is disabled and prevents it from becoming enabled. In the case that the condition is already enabled, the call fails (since conditions may not be disabled).

The way that the condition machine ensures that the condition will not become enabled is simply by locking the machine. The caller must call the unlock function when they are done performing their access. For this reason, all accesses under this function must be very short and passing instance data back to the user is precluded.

The other case, of course, is when a function requires that a condition be enabled in order for an access to succeed. It uses the call `g_variant_require_condition()`. The call ensures that the condition is enabled (taking steps to enable it, if necessary).

In this case, no locks are used. The one-way nature of conditions provides a guarantee that accesses requiring the condition to be true will now be safe for as long as the life of the instance.

The conditions (and their directions) were chosen with this in mind. When the condition machinery is not running, the places where a condition is forbidden are few in number (three) and extremely short in running time (never more than one or two dereferences plus associated refcounting).

For this reason, contention is very low during read accesses.

The fact that there are a small finite number of conditions that can be enabled and that all access is lock-free once they are enabled also means that the total amount of contention is kept quite low.