

Nidificate Programmer's Reference Guide

License

Nidificate is © Copyright 2002 by Lewin A.R.W. Edwards. It is released subject to the following BSD-esque license:

Redistribution and use of the Nidificate documentation and program in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the disclaimer below.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the disclaimer below in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features provided by this software must display the following acknowledgement: "This product includes software developed by Lewin A.R.W. Edwards".
4. The name of the author (Lewin A.R.W. Edwards) may not be used to endorse or promote products derived from this software without specific prior written permission. Compliance with point 3 above is not considered to be endorsement or promotion.

Disclaimer: Nidificate is supplied free of charge on an "as-is" basis, and no promise or guarantee of functionality or technical support is provided. By using this product, you take full, unlimited responsibility for any loss or damage caused to you or others as a result of using the product, and you agree to indemnify the author from any such loss or damage. You further acknowledge that the author is under no obligation to answer questions, provide assistance or modifications, or otherwise supply ongoing technical support for Nidificate or for other products that employ Nidificate.

To summarize the above: Nidificate is free to use, to examine, to modify and to distribute, as long as you acknowledge my authorship, you don't expect technical support from me, more important yet you don't tell your customers to request technical support from me, and you accept full responsibility for anything you make my code do to the world. Of course, on principle I encourage you to make your project open source, but I don't require that you do so.

Introduction

When implementing embedded operating systems, particularly operating systems that include a graphical user interface, there is frequently a need to develop external media objects, such as bitmapped image data for icons and window controls, and then incorporate these objects into the final binary image. It is convenient to work with these objects on the development platform with readily-available tools using common file formats such as Windows BMP, and convert the final result to space-efficient/time-efficient formats as a penultimate, one-way step prior to generating an executable firmware image.

In some embedded systems, it may also be useful to be able to switch out the entire set of resources in one step, or allow the user to change many resource selections simultaneously using a single menu selection. An obvious example application would be to allow the user to select a “skin” for the operating system, which would include typefaces, icons, and other appearance data. Another typical example would be to allow installation, by the end-user, of an international language pack to permit a device to display content in a foreign language. Using replaceable resources in this way also permits a product to be internationalized by a third party without the need to release proprietary sourcecode or involve the third party directly in the build process. It can also shorten the development cycle on some embedded systems, by enabling the developer to upload only the code component of the OS when resources have not been updated.

Nidificate is a combination of a host-side archiver and format conversion tool (Nidify), and a library for the embedded system that allows your embedded OS to access resources inside a proprietary archive file (referred to as a “nest”). The host-side archiver was developed with Microsoft Visual C++ 6.0 for Win32, but it uses only standard file I/O calls and can easily be ported to any operating system. The device-side library is designed to work on any platform for which a properly functioning C compiler is available, though it is specifically intended for use with fairly “meaty” 16/32-bit platforms (I have used it on 68000, ARM, PA-RISC and Intel x86). The existence of a filesystem is *not* assumed; it is intended that the nest file be mapped directly into the processor’s address space, either by a (user-supplied) function that loads it into RAM from a storage device, or by including the nest file at a known address in flash or ROM; the nest file itself becomes a read-only memory-mapped filesystem of sorts. Issues such as endianness and integer size are handled implicitly by the file format (also, it is not necessary to enable structure packing/disable word boundary alignment in your compiler, since the nest file is accessed bitwise at all times). All sourcecode is supplied for the user’s convenience.

As an additional feature, the archiver tool can easily be extended to convert input files from the format you find convenient on the host platform, to formats convenient for the target platform. The current version supports the following conversion methods:

- 24-bit Windows BMP to a proprietary internal 5:6:5 or 8:8:8 RGB data format suitable for easy integration with simple GUIs.

- A text resource list similar to a Windows stringtable or Macintosh STR# resource. The current version of this library only supports one text resource per nest file, with a maximum of 2^{32} strings. String handling is especially optimized for speed on the embedded platform.
- Raw binary (include with no conversion).

The archiver works using a simple input scriptfile, and creates both the nest file and a C header file that defines resource IDs for the file(s) in the nest. You may generate the nest file as a binary image, or optionally also as a C header file for direct inclusion in your program.

Using Nidify to Create Your Archive

Nidify is the archive-creation utility which you run on your development system. It takes as input a plain-text scriptfile (.nid file) and one or more resource files (.bmp, .bin, etc) and generates an output nest file (.nst) with an associated C header file containing ID number definitions for your resource. (As well as the binary nest file, you may choose to output the nest to a C header for simple inclusion into your program. The two output methods have different advantages; it's simple to load the included version, because you know exactly where it is already, but the binary version can more easily be kept separate from the main code mass).

Nidify takes a single required argument, *filespec*, which is the path and name of the scriptfile to be processed (e.g. *filespec.nid* or *c:\path\filespec.nid*; the filename is not important but it MUST end with an extension, and .nid is recommended). The output files (*filespec.nst*, *filespec.h* and, optionally, *filespec_bin.h*) are created in the same directory as the scriptfile.

If you also wish to create the optional C header version of the nest file, add the switch "-h" to the commandline.

Note that all non-absolute filespecs in the scriptfile are considered to be relative to the *current* directory. For this reason it is recommended to run nidify from within the directory containing your scriptfile and media files.

The scriptfile consists of an arbitrary number of lines of the form:

text-id,resource-type,resource-param

where:

- *text-id* is a C-style macro name that will be defined in the generated header file to match the numeric resource ID assigned to the resource data. Nidify numbers all resources of a given type (BINARY, BMP565, BMP888, etc) in ascending order, and generates a header file reflecting the automatically-generated numeric assignments for the text IDs you specify in the scriptfile. If you change the order of the lines in the scriptfile, you may also change the numbers assigned to each text ID after the change; this situation would require you to recompile your application.
- *resource-type* identifies the type of the resource. This affects input translation methods and the resource tag specified in the output nest file. Allowed (case-sensitive) values for this field are:
 - BINARY** – the file specified by *resource-param* is included in the output nest file as a resource of type NEST_RSRC_BINARY, with no translation.
 - BMP565** – the file specified by *resource-param* must be a 24bpp uncompressed Windows bitmap image. It will be translated to a 5:6:5 raw bitmap format (refer to "Supported Resource Types and File Conversions) and stored as a resource of type NEST_RSRC_BMP565.
 - BMP888** – the file specified by *resource-param* must be a 24bpp uncompressed Windows bitmap image.

It will be translated to an 8:8:8 raw bitmap format (refer to “Supported Resource Types and File Conversions) and stored as a resource of type NEST_RSRC_BMP888.

STRING – The user string specified in *resource-param* is placed into the NEST_RSRC_STRING resource. *resource-param* must be enclosed by quotation marks (“...”). The string may contain any ASCII character from 32-255 inclusive; Nidify identifies the string as starting at the first character after the first quotation mark after the second comma on the script line, and ending at the last character before the last quotation mark on the script line. The string may also contain the following case-sensitive escapes:

\\	Backslash
\b	<BS>, 0x08
\t	<TAB>, 0x09
\n	<LF>, 0x0A
\f	<FF>, 0x0B
\r	<CR>, 0x0D
\e	<ESC>, 0x1B

Unknown escapes will be ignored silently. Any illegal characters (0-31) will be stripped out and will not be emitted to the nest file.

Before Nidify begins processing each script line, whitespace (characters from 0-32 inclusive) at the beginning or end of the line will be stripped. Any empty line, or any line that begins with the character “#”, is considered a comment and ignored.

By way of example, the following script:

```

#
# My important project, version 1.00
#

ID_MYRESOURCE,BMP565,mypicture.bmp
ID_MYBINARY1,BINARY,binary1.bin
ID_MYBINARY2,BINARY,binary2.bin
ID_PICTURE,BMP565,picture2.bmp
STR_WELCOME,STRING,"Welcome!"
STR_EMPTY,STRING,""
STR_QUOTE,STRING,""My my," said Fred"

```

will produce a header file that looks like this:

```

/*
    Nidification resource header file
    AUTO-GENERATED by Nidify (v1.0) - DO NOT MODIFY
*/

/* Stringtable IDs */
#define STR_WELCOME 0      // Welcome!
#define STR_EMPTY 1      //
#define STR_QUOTE 2      // "My my," said Fred

/* Non-string resources */
#define ID_MYRESOURCE0    // (BMP565) mypicture.bmp
#define ID_MYBINARY1 0    // (BINARY) binary1.bin
#define ID_MYBINARY2 1    // (BINARY) binary2.bin
#define ID_PICTURE 1      // (BMP565) picture2.bmp

```

Usage suggestion: To take best advantage of the amenity granted by the nest file format, I suggest you structure your scriptfile: separate it into sections for each different type of resource you use, and do not modify the order of the lines within a section (reordering the sections themselves is allowed). This way, the ID generated for each resource will remain constant across nest build runs, and by following these simple rules, you will ensure that regenerating the nest (e.g. to change the graphical appearance of your application, or translate it into another language) never requires you to recompile your application code.

Supported Resource Types and File Conversions

Binary Any desired custom data can be stored in a binary resource (NEST_RSRC_BINARY). The nest file will contain a raw import of the user-supplied data file.

BMP Nidify supports conversion of 24-bit Windows BMP data to either 5:6:5 or 8:8:8 RGB stored in resources of type NEST_RSRC_BMP565 and NEST_RSRC_BMP888 respectively.

The first four bytes of the BMPxxx resource are the image width in pixels (not bytes), in big-endian format. The next four bytes are the image height in scanlines, in big-endian format. This information is followed immediately by raw pixel data in intuitive “page reading order”; i.e. the upper-left pixel is first in memory, and the lower-right pixel is last. (A Windows BMP file is normally stored in a strange bottom-up scanline order that is inconvenient to work with; Nidify hides this eccentricity from you).

When converting to a BMP565 resource, the pixel data is formatted as RRRRRGGGGGBBBBB in big-endian format, i.e. the first byte in memory contains RRRRRGGG, the next byte contains GGGBBBBB for the same pixel, and so on.

For BMP888, the pixel data is formatted as 8:8:8 RGB (RRRRRRRR, GGGGGGGG, BBBBBBBB).

Nidify performs some basic checks on the input BMP file for a BMP565 or BMP888 conversion; it validates the “BM” signature in the first two bytes of the file, the least significant byte of the header length (offset 2 in the file, which should be 0x36) and the least significant byte of the format type (offset 0x0e in the file, which should be 0x28 for uncompressed RGB).

STRING Nidify supports creating a single string resource of type NEST_RSRC_STRING. This resource consists of an arbitrary number of user strings, stored as ASCIIZ data with a double zero terminator. The special string 0X01, 0x00 is used to represent an empty string. Strings are identified by their relative position in the resource; e.g. in a string resource containing “A\0BCD\0Fred\0\0”, string ID#2 is Fred.

Note that the nest file format is capable of supporting multiple string resources, but the supplied Nidificate library and Nidify utility only support generating a single such resource per nest. For performance reasons, Nidify places the string resource, if any, at the very start of the nest.

API Reference - Using The Nest in Your Embedded Application

Simple nest-parsing functionality and required constant definitions are included in the files nest.c and nest.h; these are the only files you need to move across to your embedded project (apart from the resource ID definition header file for your nest, and the nest itself). Accessing resources within the nest is accomplished using the following function calls (defined in nest.c). These functions assume only that you have already loaded the nest into your processor's address space somehow, either by including it directly in your C program, by loading it into RAM (e.g. from a filesystem) or by including it at a known location in flash or ROM.

For definitions of the error types reported by this version of Nidificate, refer to nest.h.

NESTERR Nest_Validate(NESTHANDLE nest)

Verifies that the handle refers to a valid nest file, and that all resources in the nest are intact. This function checks the nest signature, version information (any nest file major version number equal to or less than the major version number of the library being used is acceptable), and verifies checksums for all resources in the nest. This function is potentially time-consuming; it is intended for use in situations such as validating user-supplied nest upgrades. A typical such application would load the new nest into RAM, validate it with Nest_Validate() to ensure structural integrity, and then fetch an application-specific string or binary resource from the nest to check that the file data is versioned correctly for the product being upgraded.

const unsigned char *Nest_Get_Binary(NESTHANDLE nest, unsigned int rsrcID)

Returns a pointer to the start address of the requested BINARY resource, or a null pointer if this resource cannot be found.

const unsigned char *Nest_Get_Binary(NESTHANDLE nest, RSRCTYPE rBmpType, unsigned int rsrcID, unsigned int *width, unsigned int *height)

Returns a pointer to the start address of the requested bitmap (BMP565/BMP888) resource, or a null pointer if this resource cannot be found. On entry, rBmpType must be one of NEST_RSRC_BMP565 or NEST_RSRC_BMP888. If width and height are non-NULL, their destinations will be set to the bitmap dimensions on exit. (In embedded applications, the size of the source bitmap is frequently known already, dictated by screen layout or other considerations. In such applications, your device-side code probably has a macro defining a few standard sizes, so you will not care about the size fields in the bitmap resource).

const char *Nest_Get_String(NESTHANDLE nest, unsigned int stringID)

Returns a pointer to the ASCIIZ data of the specified string ID, or a null pointer if this string cannot be found. This function performs little error checking; it searches for the first STRING resource in the nest and locates the stringID'th item in the resource.

Nest File Format

The nest file is led-in by an ID/versioning header consisting of the following data:

- 4 BYTES - 0x4E, 0x45, 0x53, 0x54 (“NEST”).
- 1 BYTE - Major version of the Nidify utility used to generate the nest file. Files with a given major version number can be read by all library versions with the same or higher major version number.
- 1 BYTE - Minor version of the Nidify utility used to generate the nest file.

This header is followed by an arbitrary number of structures of the following format:

- 1 BYTE – Resource type code, which should be one of the NEST_RSRC_* constants defined in nest.h. The special value NEST_RSRC_END marks the end of the nest file; if the parser encounters this resource ID marker, it stops seeking immediately, and it is safe to omit all of the following fields in the end-of-file resource.
- 1 BYTE – Simple checksum of resource data area.
- 4 BYTES – Size of this resource’s data area (i.e., excluding this header), in big-endian format.
- 2 BYTES – Resource identifier, in big-endian format. This identifier must be unique within a given resource type; i.e. it is legal to have a BINARY resource numbered 1234, and a BMP888 resource also numbered 1234, but it is **illegal** to have two BINARY resources numbered 1234. Nidify cannot generate an illegal nest.

Resource data is formatted internally as described in “Supported Resource Types and File Conversions”, above.

History and Future Enhancements

Code in Nidificate dates back to work I originally carried out in 1989 on the Commodore Amiga. At the time I was trying to write a generic game engine, but the code I developed inevitably turned into at least the conceptual basis for almost all of the GUI projects I've been involved in, both in software for desktop computers and firmware for embedded systems.

Nidificate is a sanitized version of that old Amiga code, mixed with things I've learned (or simply needed to develop) over the years between 1989-2002 including home-made games, consumer appliances, and ports of arcade game emulators to arcane systems. The GUI code I have here in my archives is spread across about a dozen projects, and it supports, among other things:

- Huffman-compressed resources.
- DES and RSA-encrypted resources.
- Signed resources.
- Dialog boxes with definable geometry, colors, title text, and controls (sliders, spinwheels, radiobuttons, checkboxes, list boxes, combo boxes, text-entry controls and pushbuttons).
- Fixed and proportional-space font rasterizer with scaling.
- Various drawing functions (rectangle, ellipse, rounded-corner rectangle, line, etc). for 1-bit, RGB 4:4:4, 5:6:5, 8:8:8 and 8-bit palettized video devices, with interesting features such as implicit rotation (for situations where the RAM layout to physical on-screen layout of pixels is abnormal or even changes at runtime).
- Full windowed GUI type features with application-transparent clipping, buffering of invisible window data and so on.
- Sprite functions with scaling, mirroring and rotation, intended for game applications.

It is my intention to extract as much of this functionality as possible from my ancient projects, clean it up to fit within a single set of terminology conventions, and document it fully. The end goal is to provide embedded systems developers with a useful, free GUI library and associated host-side development tools. Getting Nidificate clean, sane and fully documented was a required first step; embedded GUI features to come will rely on the nest as their basic resource format.