Dynamic Software Update on Embedded Systems

A. Mileto

April 22, 2023

Contents

1 Project Data **2**

2 Project Description **2**

2.1 Design Implementation 2

3 Project outcomes **2**

3.1 Concrete outcomes 2

3.2 Learning Outcomes 3

3.3 Existing knowledge 3

3.4 Problems encountered 3

1 Project Data

- Project supervisor: Daniele Cattaneo
- Person delivering the project

Last and first name	Person code	Email address
Alessandro Mileto	10868033	alessandro.mileto@mail.polimi.it

2 Project Description

The topic of the project is the investigation of Dynamic Software Update on Embedded Systems software with the support of a suitable compiler. This project sets the ground for such a compiler to be developed. In particular, an application is ran and its code is dynamically substituted preserving its state. This way, update strategies could be easily tested and developers are provided with some hints on how to design backend logic of the future compiler. The application running is a very simple one, Tic Tac Toe whose state is the content of the game board.

2.1 Design Implementation

The application chosen to carry out the experiments is Tic Tac Toe. This latter is the core part of the project and it has been distributed as a GNU C shared library which is the **updatable** module of the application. The structure of the library, apart from game-specific routines, is described in the header file updatable.h. The routine loop() is the application library main function, while init() and destroy() are added for flexibility and ease of initialization/destruction of the shared library object. The main loop executes the application specific routines (such as game actions, printing the board etc.) , among which **stopping points** are located. They are the pivotal part of the updatable application infrastructure: whenever a given action is executed (such as printing the game board, in the specific case of this project) the application checks its state at each stopping point and, eventually, prepares for updates if any. The macro STOPPING_POINT marks the above mentioned hooks. It basically extends to a check of the HALT flag. HALT and UPDATE are two flags defined in the updatable.h header which help realize dynamic update: when HALT is set to 1, loop() breaks and the UPDATE flag is set to 1. It means that the library is ready to be swapped.
The application is built as a shared library because it is intended to be used with the Dynamic Loading APIs. In fact, Unix-like OSes provide support for opening the library, looking up symbols (functions, variables etc.) and eventually calling them. This is what the main.c file does. It loads the library and starts the loop() function using the dlopen() and dlsym() routines (see GNU Linux Manual).
Besides that, main.c installs a custom handler named **update_available** for SIGUSR1 (user defined signal, see GNU Linux Manual using *man 7 signal*): it notifies the used an update is available and sets the HALT flag to 1 (see above). This is done in order to simulate the *update available* event. Upon signal receipt, the main loop of the library application stops. In turn, the apply_update routine opens an update facility (another shared library supposed to be distributed together with the library update itself) and executes the **state_change_transformation** function in it, which takes two pointers (two handles, one of the old version of the library and the other of the new version) and performs an update-specific action to change the old state into the new (upon the eventual initialization of the new library). The author of the update is in charge of writing a suitable update handler every time.
If the update is successfully applied, control is given back to the main which resolves the new loop() symbol and restarts the application.

3 Project outcomes

3.1 Concrete outcomes

First of all, the project resulted in the coding of a playground where to test and devise embedded application hot swap logic: this enables developers to test in a toy environment their patches before production and seamless integration. Secondly, this paves the way for the development of a compiler capable of including in a given application respecting the (rather flexible) structure defined in updatable.h the utilities and back end data structures needed to enable dynamic updates. Finally, even if update handling is a rather flexible task in this setting (as the handler is a shared object written by the patch provider) it may be that the future compiler will be able to produce the update handler itself along with the patch. In this latter case, the playground may help the development of automatic update handling.

3.2 Learning Outcomes

I learned a lot in terms of system programming, IPC and library management. For the first time in my life I used the DL API (I was not even aware of them) and, I think I will be using them again and again. Moreover, I coded a makefile for the first time, which is a rather simple task but I had never relied on manual solution for task automation as I had always used IDEs or automatic build toolchains.

3.3 Existing knowledge

The topic has been largely investigated in the past years and several research articles have been published in various venues. Both [3] and [1] convey some core ideas about the updatable application being a big loop, where stopping points have to be placed in order to realize the update. [3] points out strongly that update points have to be safe, suggesting that compiler authors have to take extra care at implementing this kind of functionality. Moving on a more practical ground, [2] explains a practical methodology on how implement seamless, state-safe and efficient dynamic updates. The authors provide an example of application of such methodology to a compiler suite even focusing on the automatic patch generation. Last but not least, [2] points out a strategy to implement simple playgrounds as the one this project is about, id est the DL APIs.

3.4 Problems encountered

Most of the problems encountered have to do with the Dynamic Loading APIs. In particular, errors in symbol retrieving were among the most frequent bugs I happened into.
First of all, when a shared object is opened using *dlopen()*, some flags are used to control symbol resolution (e.g. lazy resolution) and visibility (e.g. the symbols of the loaded object may or may not be visible to subsequently loaded shared objects). It turned out that if a shared object is opened with the **RTLD_GLOBAL** flag, when the update handler looks up the symbols of the open shared libraries (the new and the old one), name clashes can hamper symbol resolution. To be more precise, let us suppose two symbols have the same name (e.g. the state of the app libtictactoe.c is name x in both the initial and updated version of the application of the project). When such a clash happens, symbol resolution is carried out in a particular order (see *man 3 dlopen* and *man 3 dlsym* for further details) which depends on the configuration of the environment. In my personal case, the handler failed at resolving the old symbol **x** after the updated shared object was open because the more recent x definition (contained in the updated library) had priority over the less recent one even if I performed the dlsym operation passing to the function the old library handle. The flag **RTLD_LOCAL** solved the problem in my case.
Another tricky bug is related to symbol resolution itself, with no name clashing. When programming the logic of the application, extra care has to be taken that all symbols are correctly resolved before calling them (it refers to functions/macros etc.). Not taking care of resolving new version of the symbols after the dlclose() and dlopen() is the best way to trigger tons of segmentation faults due to the dereferencing of null pointers or pointers referring to the virtual address space of the closed shared object: sometimes this is hard to debug because, for instance, it may be that symbols are called in loops (see main.c, where the loop() function has to be resolved each time the loop is restarted).
To sum up, extra care has to be taken when dealing with symbol resolution, not only in case of name clashing.

References

[1] Michael Hicks and Scott Nettles. “Dynamic Software Updating”. In: *ACM Trans. Program. Lang. Syst.* 27.6 (Nov. 2005), pp. 1049–1096. ISSN: 0164-0925. DOI: 10.1145/1108970.1108971. URL: https://doi.org/10.1145/1108970.1108971.

[2] Iulian Neamtiu et al. “Practical Dynamic Software Updating for C”. In: *SIGPLAN Not.* 41.6 (June 2006), pp. 72–83. ISSN: 0362-1340. DOI: 10.1145/1133255.1133991. URL: https://doi.org/10.1145/1133255.1133991.

[3] Min Zhang, Kazuhiro Ogata, and Kokichi Futatsugi. “Towards a Formal Approach to Modeling and Verifying the Design of Dynamic Software Updates”. In: *2015 Asia-Pacific Software Engineering Conference (APSEC)*. 2015, pp. 159–166. DOI: 10.1109/APSEC.2015.28.