

# *Embedded System Update*<sup>1</sup>

Jürgen Salecker  
Siemens AG, CT SE 2  
Otto-Hahn-Ring 6, 81730 München, Germany  
[juergen.salecker@siemens.com](mailto:juergen.salecker@siemens.com)  
V 1.4.1

The *Embedded System Update* architectural pattern applies to distributed embedded systems which require an update on a regularly basis. This pattern ensures that messages like “please do not switch off power during flash programming” are not necessary any more, because the patterns provides a solid and robust update mechanism, even in cases of power faults during system upgrade and erroneous software within the update.

## **Context**

This pattern applies for distributed embedded systems which require a robust solution for system upgrades.

It requires that the embedded system has a non-volatile memory storage area that is large enough to hold two version of the software at the same time. In cases where this non-volatile storage area of the embedded devices is not sufficient for keeping two images the image might be incrementally split into smaller components so that they fit into the available non-volatile memory storage area.

## **Example**

Consider a DSL router located in the cellar of your house, which needs to be updated. The PC is in the office room and the update is started, then usually a message like “please do not switch off power during info LED blinking” appears. This informs the user that a critical transaction is ongoing. But details like that the device is in the cellar, the update initiator is not able to see any LED blinking, because he is just too far away, further complicates the situation. Let’s assumes at the same time a fuse blows up and switches

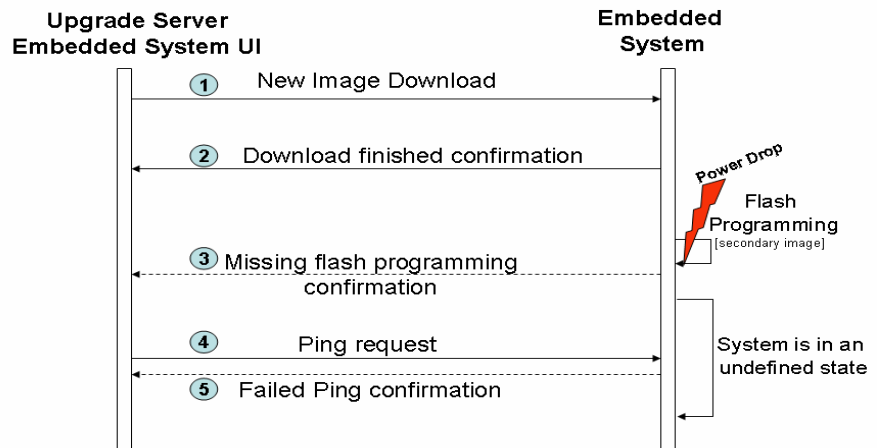
---

<sup>1</sup> © Siemens AG 2006, all rights reserved; permission is granted to EuroPLoP to make copies for conference use and to publish it in the official conference proceedings.

off the power for the house. In this case you might be left with an uninitialized router and in addition you might not be able to get easy a replacement, because you are not able to reconnect the router and you are also disconnected from the internet.

**Problem**

There are basically two main problems what can happen during a download, a power drop during flash programming or erroneous software within the upgrade. What happens if the power drops during flash programming is illustrated in the figure below:



*Figure 1, Power drop during flash programming*

Due to the fact the power drop happened during flash memory programming this memory will be left in a partly initialized state. The consequence is that the embedded system is not able to start-up as expected. Even worse the system is left in an undefined state and without physical access a repair is impossible.

A fatal software error within the update image might force the embedded device into a similar situation as with the power drop during flash programming. The figure below illustrates this scenario:

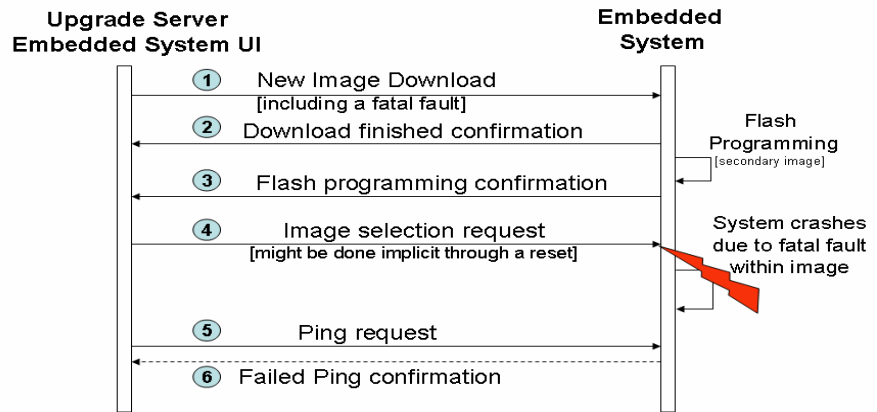


Figure 2, Erroneous software within the image

After the new image - with the erroneous software - has been selected the embedded system will crash and the system will be left in an undefined state. In both cases an embedded device - that does not use this pattern here - will be left in an undefined and therefore not deterministic state afterwards.

This raises the following question: “How would it be possible that an upgrade of the embedded system converges always towards a stable state?”

Where a stable state is defined that either the new upgrade is operational and at least ready for a new download or in any other case no permanent changes on the embedded system should have happened.

The solution for this problem is influenced by the following forces:

1. An unsolicited event – like a power fault - during the upgrade process, especially flash programming will leave the embedded system in an undefined state. Because only a part of the image or component has been stored this might prevent the embedded system to start-up again.
2. For economic reasons it does not make sense to provide the embedded system with a 100% reliable power supply, or this might be even physical impossible.
3. The software upgrade could contain a programming error which causes a malfunction in parts of the embedded system or shuts down the embedded system completely.

4. The customer should be able to perform an audit on the upgrade, in such a way that he could switch back to the previous version, if he is not 100% satisfied.
5. The embedded systems are difficult to access, or introduce a potential physical danger to the people involved. In those cases it has to be prohibited by software architecture that a software update is getting influenced by external forces in such a way that a permanent malfunction of the embedded system is the result. Thus enforcing a potential dangerous human action, because the embedded system has to be replaced.

## Solution

Provide enough flash memory to hold two software images within the embedded device: a primary and a secondary image. In normal operation the primary image is the software that executes and the secondary image is overwritten by an update operation. A confirmation from a user is required to decide whether the new secondary image might become the new primary one. If this confirmation is either negative or is not provided at all, nothing will change and the original image will stay as the primary image.

The secondary- and the primary-image are associated with following attributes:

Attribute	Values	Description
<i>signature</i>	a number	Unique hashcode or a signed hashcode (a real signature) of this image.
<i>state</i>	<i>primary</i> / <i>secondary</i>	The value <i>primary</i> defines the images which has been validated and <b>is</b> currently in operation. The one with the value <i>secondary</i> defines an image which is currently <b>not</b> in operation.
<i>launched</i>	<i>[-1..n]</i>	An integer number which counts the number of start-ups executed. A <i>negative number</i> defines a non existing image. A <i>null</i> defines a new image which has not been launched yet. A positive number defines the number of times this image has been tried to launch.

A bootstrap loader on the embedded device decides which image should be launched after system reset. An image will contain the attribute *primary* if the initiator of the download did provide a “confirmation of correct operation” for this image. The primary image is the default image to be launched on the embedded system. If available the secondary image will be launched exactly one time – after reset - on the embedded system.

Any time a new upgrade is downloaded to the embedded device it becomes the state *secondary*. After the download has been finished, the secondary image will be launched for execution, either via user request, or implicit via system reset on the embedded device

An image becomes the state *primary* exclusively if the initiator of the download did “confirm the correct operation” of this image, in any other case nothing will change. “Confirm correct operation” means that especially the part of the *secondary* image which is responsible for the download operation has to be checked for correct operation. Any other checks about correct execution, like applications tests, might be executed but are not relevant for working of this pattern.

If the user omits the “confirmation of correct operation”, automatically the previous *primary* image will be launched with the next system reset because a secondary image will be launched exactly one time. After this operation the system is in the state where it was before the download operation.

This holds as well if the download upgrade did contain a fatal programmer fault, causing the system to crash during secondary image launch. In this case the “confirmation of correct operation” is missing and the system goes back to the *primary* image, into the stable state, where it was before the upgrade request.

## Structure

The principle structure of the solution is shown in the picture below:

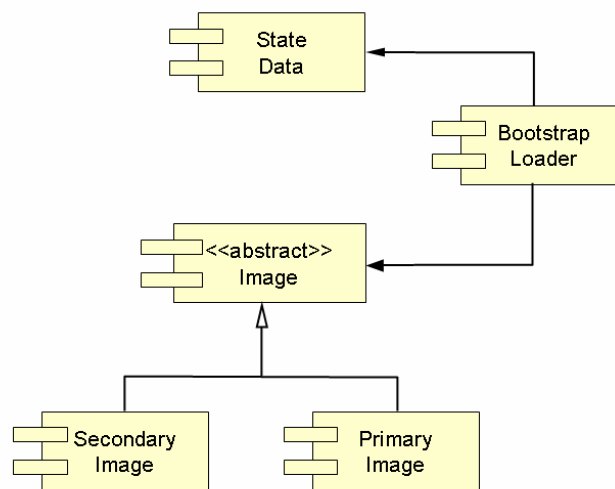


Figure 3, Embedded System Components

Based on the state data the bootstrap loader selects the appropriate image for execution, which could be even the bootstrap loader itself.

**Implementation** The principle operation of a safe upgrade operation is illustrated in the sequence diagram below:

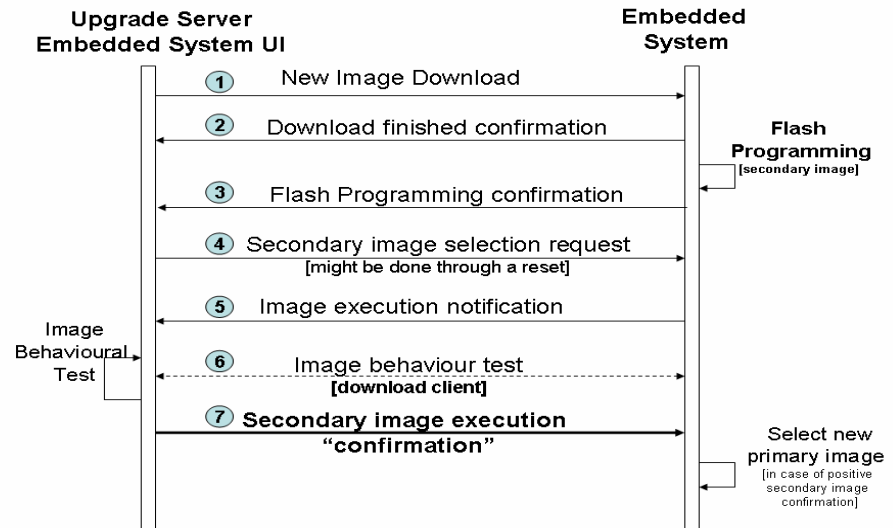


Figure 4, Safe Embedded System Update

The most important operation in the above sequence chart is action (6). Here the download server either confirms the correct minimal operation of the secondary images or rejects it. In the first case the state of the image will be changed into the state *primary* by the bootstrap loader of the embedded device.

In the second case, which is also equivalent to the case where no communication is possible between download server and embedded device, the bootstrap loader will select the previous, i.e. primary image on next system startup automatically.

### **Minimal Operation Check for Secondary Image**

There is one essential operation required within the new upgrade, it has to be made sure that – under all circumstances - another download is possible again. Further tests might be executed, for example performing an audit and optional rejecting or accepting this update, but additional tests are not essential as long as another download operation can be executed.

The download server has to initiate a build-in test in the download client of the image that again a download is possible. Only in the case where this test returns satisfying results the download server might confirm the correct operation of the secondary image.

### **Image Selection**

The bootstrap loader of the embedded device decides on the values of the image attributes the image to be selected, i.e. executed. It also logs image selection in the attribute *launched*.

If the size of the non-volatile memory in the embedded device prohibits the storage of a secondary image then two solutions are possible. First the image can be split into several components where every component is accompanied by a secondary component. A side effect of this splitting is a much faster upgrade speed (flash programming time) for system upgrades, as not always all components of a system needs to be upgraded which also supports much faster turn-around speed for system debugging and development. This again might be a significant help for faster “time-to-market”.

Second the introduction of a multi-stage bootstrap loader (-> Pattern Dietmar Schütz) which “bootstraps” the image from a secondary storage (download server) area. -> eventually a 2<sup>nd</sup> pattern.

**Known Uses** In the Q-Marine system (a seismic recording system) from Schlumberger/WesternGeco the pattern is used for managing the software upgrades of all embedded devices (several hundreds devices per system) in order to take

into account safety issues, i.e. avoiding unnecessary hardware exchange by people.

Embedded systems for space missions might have implemented this pattern as well; however I do not have any detailed information about this.

The telecom industry might have implemented this pattern as well (Siemens COM), for managing the software upgrades of phone switches.

**See-Also**

In order to have always a well-known and valid combination of components the BOM pattern (EuroPlop 2004) is recommended to be used as an underlying construct.

**Consequences** The pattern provides the following **benefits**:

It provides a fault tolerant and safe update method for embedded systems and reduces therefore the stress level for the operator of those embedded devices.

Unsolicited external events – during the update process - do not have any relevant impact on the operation of the embedded system.

The pattern has the following **liabilities**:

The size of the non-volatile memory (typical flash memory) within the embedded device has to be twice as large as for one image.

The required additional flash size might limit the use of the pattern for very low cost devices.

Additional time and resources for testing is required.

**Credits**

My former colleagues at Schlumberger/WesternGeco Cyrill Camiul and Hermann Pehl who did main parts of the implementation and concept development. My shepherd Peter Sommerlad did provide significant help for the “implementation” of this pattern into writing.