F&S Update Tool

CheckAutoStart and Update
Windows Embedded Compact

Version 1.03
2015-08-05

Untere Waldplätze 23
D-70569 Stuttgart
Fon: +49(0)711-123722-0
Fax: +49(0)711 – 123722-9

History

Date	V	Platform	A,M,R	Chapter	Description	Au
22-09-2014	1.01	All	M	*	Changed to New Company CI	JG
18-05-2015	1.02	All	M	*	Minor layout corrections	HF
07-05-2015	1.03	All	M	History	Added new remark regarding Compact 2013	HF

V	Version
A,M,R	Added, Modified, Removed
Au	Author

Remark

In the remaining document we'll use the term "Windows CE" as generic reference to Windows Embedded CE and Windows Embedded Compact.

Remark

The name of the application is Update.exe (Windows Embedded CE 6.0 and Windows Embedded Compact 7) or Update2013.exe for Windows Embedded Compact 2013.
In the remaining document we'll use the term "Update" as generic reference for both applications.

Remark

In the remaining document we'll use the term "NetDCU" as generic reference to all our Windows Embedded CE and Windows Embedded Compact boards. This should also include armStone™, efus™, PicoCOM, PicoMOD and QBliss boards where appropriate.

Table of Contents

1 Introduction

Many applications involving the NetDCU don't require any input device like a mouse or keyboard. And updating the software installed on these NetDCU boards in the field is often not so easy. You most probably have to use a portable computer, write some special update software and then have the responsible technician doing the update on location.

The main goal of our update software was to make this procedure as easy as possible. In fact our idea was to use a portable storage device, like USB Memory Stick, SD-Card or a PCMCIA Compact Flash Card (depending on the capabilities of the NetDCU), plug it in the running board and then the updating takes place automatically. And if anything fails during the update, the original state with the previous software is restored again, so that the continuous operation of the board is guaranteed. This is so easy that even untrained personnel is capable of doing it with a few simple instructions.

However when designing this concept, we encountered a problem that we had to solve first.

When the NetDCU board is already running when we plug in the update storage device, then the main application is also currently executed. This means that all the files we want to update may be under access. For example the `.EXE` file of the application itself is locked and can't be exchanged then.

On the other hand if we plug in the update storage device when the board is switched off and then start the board, it takes some time until Windows CE recognises the device – up to 20 seconds or even more, depending on the device type. During this time the main application again has enough time to start and we end up in the same situation as before.

1.1 The Two-Stage Update Process

This brought us to the idea of using two stages for the update.

Stage 1:

Plug the update storage device into the NetDCU board. The device including the `Update` program is recognised and a boot delay is set.

Stage 2:

Reboot the board. Now the boot delay takes place and prohibits the main application to start. During this time the update storage device is recognised again and the `Update` program is started. Now the update can take place.

So not only the update process is divided into two steps, also the software has been split in two parts: on one hand a resident part on the NetDCU, doing the boot delay and checking for the update storage device, and on the other hand the `Update` program on the update device itself, doing the actual file replacement.

We called the resident part `CheckAutoStart`, as it can start any application automatically, not only the `Update` program.

2 Check for Auto Start Program

This program is already included in newer NetDCU kernels. It provides two functions:

1. Delay the boot process by the `BootDelay` value set in the registry. After the delay is over, let the main application start.
2. By working silently in the background, check when new storage devices are introduced into the system. If there is a directory `NetDCUx` on this device and a program `Auto-Start.exe` in there, start this program immediately.

`CheckAutoStart.exe` is started via the registry key

\HKEY_LOCAL_MACHINE\init

This key tells Windows CE which programs are started automatically during the boot process and in which sequence. Every program is defined by value pair, `Launch<n>` where `<n>` is a decimal number defining the sequence. The higher the number, the later the program is started.

However Windows CE does not wait until the program is finished before starting the next one, because most of these programs will be running in parallel until the board is shut down. So immediately after spawning a process for one program, the next program is launched by spawning another process.

So if a program depends on the existence of another program that has to be up and running, it has to set a depend entry. A `Depend<n>` entry lists all the launch numbers of those programs, on which the corresponding `Launch<n>` (with the same `<n>`) depends. As Windows CE uses binary values for the depend entries, these dependencies have to be given as hexadecimal values with two bytes each (low byte first).

Example:

Standard start-up sequence of the NetDCU8:

Launch20	device.exe
Launch30	gwes.exe
Depend30	14 00
Launch50	explorer.exe
Depend50	14 00 1E 00
Launch60	services.exe
Depend60	14 00
Launch99	ndcucfg.exe
Depend99	1E 00
Launch100	CheckAutoStart.exe
Depend100	1E 00

This sequence starts the following programs:

1. The devices subsystem `device.exe` with launch number 20, depending on no other program.
2. The graphical subsystem `gwes.exe` with launch number 30, depending on program `0x0014 = 20 = device.exe`.
3. The Windows Explorer `explorer.exe` with launch number 50, depending on programs `0x0014 = 20 = device.exe` and `0x001E = 30 = gwes.exe`.
4. The services subsystem `services.exe` with launch number 60, depending on program `0x0014 = 20 = device.exe`.
5. The NetDCU configuration program `ndcucfg.exe` with launch number 99, depending on program `0x001E = 30 = gwes.exe`.
6. The auto start checker `CheckAutoStart.exe` with launch number 100, depending on program `0x001E = 30 = gwes.exe`.

Usually this list also shows the customer specific application launched under a number higher than 100.

Now let's examine the two functions of `CheckAutoStart` in more detail

2.1 Boot Delay

When `CheckAutoStart.exe` starts, it looks at the registry entry

`\HKEY_LOCAL_MACHINE\System\CheckAutoStart\BootDelay`

and interprets the number there as seconds to delay. This means that all programs in `\HKEY_LOCAL_MACHINE\init\` depending on `CheckAutoStart.exe` will not start until the delay is over. If this value is set to 0 or is not set at all, no delay takes place.

During this delay time, any newly detected storage device is tested for the `AutoStart.exe` program, and if found, it is started without any command line arguments. After return the boot process is continued and `CheckAutoStart.exe` ends itself.

It can take several seconds after start-up, until a removable storage device is detected by Windows CE, so just hoping that it is found before the main application starts, does not work. Therefore this artificial delay is required. It is mainly meant to avoid the start of the main application until the update storage device is recognised by Windows CE and the `Update` program can be started. Setting this delay manually is not required, as this is done automatically by the first stage of the `Update` program.

`CheckAutoStart.exe` also resets the `BootDelay` registry value to 0, so that any further reboots of the board will work again completely as usual.

Important:

To be able to delay the execution of the main application, the main application must depend on `CheckAutoStart.exe`. This means it has to get a launch number for `\HKEY_LOCAL_MACHINE\init\` higher than 100 and must list `CheckAutStart.exe` in the `Depend` entry.

Example:

The main application is called `CurrentApplication.exe`, is started as launch number 120 and depends on the graphical subsystem `gwes.exe = 30 = 0x001E`.

```
Launch120          CurrentApplica-
                   tion.exe

Depend120          1E 00
```

To make this application updateable, this entry must be changed to also list `CheckAutoStart.exe = 100 = 0x0064` in its dependencies.

```
Launch120          CurrentApplica-
                   tion.exe

Depend120          1E 00 64 00
```

Remark:

There are other methods of automatically starting an application than using `\HKEY_LOCAL_MACHINE\init\`. For example you can create a link to the `.EXE` file in the `StartUp \FFSDISK`. In fact now it is the Windows Explorer starting your application. And as you have seen, the Windows Explorer itself is started via the `init` registry key. So if you want to use this method, you have to start `CheckAutoStart.exe` before `explorer.exe` and you have to set `explorer.exe` to depend on `CheckAutoStart.exe`.

For example the following setting will do this for you: start as `Launch40` and add `40 = 0x0028` to `Depend50`.

Launch20	device.exe
Launch30	gwes.exe
Depend30	14 00
Launch40	CheckAutoStart.exe
Depend40	1E 00
Launch50	explorer.exe
Depend50	14 00 1E 00 28 00
Launch60	services.exe
Depend60	14 00
Launch99	ndcucfg.exe
Depend99	1E 00

2.2 Background Check for Auto Start

When the boot delay goes by without finding some program to automatically start, the boot process is continued. This means that any programs depending on `CheckAutoStart.exe` are now starting. This is the normal behaviour as most often the `BootDelay` registry value will be 0 and the whole boot delay step is skipped.

The `CheckAutoStart` program puts itself in the background and keeps on looking for new storage devices. A new storage device means a change in the root directory, especially a new subdirectory with a flag `FILE_ATTRIBUTE_TEMPORARY` set. So every time the root directory has directory changes, `CheckAutoStart` is notified and checks all directories having this flag set for a file `AutoStart.exe` in a subdirectory named after the platform. This platform name is taken from the registry entry `\HKEY_LOCAL_MACHINE\Platform\Name` and usually carries the name of the board, i.e. efusA9, NetDCU8, etc.

This program is then started with a command line containing

-t <n>

where `<n>` is the current boot delay (usually 0). This means the program was called after a delay of `<n>` seconds has timed out earlier.

Passing the delay time to the started program allows it to react to unexpected long recognition times. For example the `Update` program increases the `BootDelay` value by another 20 seconds, if it was not recognised in time during the current boot delay stage.

Example:

An USB Memory Stick is plugged into the NetDCU6. This creates a subdirectory "`\Hard Disk`". So the program searched for is `\Hard Disk\NetDCU6\AutoStart.exe`.

If there was no current boot delay, this program is called with command line "`-t 0`". If the current boot delay was 20 seconds, but it went over without detecting any AutoStart program, it is now called with command line "`-t 20`". And so on.

Please note:

Every time there is a change of subdirectories in the root directory, all subdirectories having the `FILE_ATTRIBUTE_ TEMPORARY` flag set are searched again, no matter if they existed already before or not. So continuously creating and deleting directories directly under `\` is no good idea, as it triggers this search every time. In real life we don't expect this to be a problem. Creating normal files in `\` also doesn't matter.

After `CheckAutoStart` has executed the `AutoStart.exe` program, it really ends itself. So from then on, newly plugged in storage devices will not automatically be recognised anymore. This is to avoid triggering the same `AutoStart.exe` program twice when other directory changes happen in the root directory.

3 The Update Program

As we have seen before, the `Update` program works hand in hand with the `CheckAuto-Start` program. It is automatically started by `CheckAutoStart` and therefore must be named `AutoStart.exe` and has to reside on the removable storage device used for the update procedure. However we'll still call it the "`Update` program" throughout the rest of this document to avoid confusion with any other `AutoStart.exe` program that may be present.

There are two situations when the `Update` program can be called:

1. When the board is in normal operation mode. Then we'll get a command line "`-t <n>`" where `<n>` specifies a timed out boot delay. Then we can do nothing but set a new `BootDelay` value and ask for reboot.
2. When the board is in the boot delay stage. Then we'll get an empty command line and we know that the update procedure can take place now.

Let's look at these two situations in more detail.

3.1 Update Program With Command Line "`-t <n>`"

The board is in normal operation mode. Especially the main application is running. We can't do the update as some of the files to replace might be locked. This is the first stage of the two stage update process.

In this case, the `Update` program simply increments the delay value given on the command line by 20 seconds, stores this value in the registry entry

`\HKEY_LOCAL_MACHINE\System\CheckAutoStart\BootDelay`

and shows a dialog box requesting the user to shut down the current application and reboot the board. If the dialog box is closed, the `Update` program ends itself.

3.2 Update Program With Empty Command Line

When the `Update` program sees an empty command line, it was either called manually, or during the second stage, the boot delay stage, of the two stage update process. Then the actual update procedure may start.

The update is controlled by an update script that has to reside in the same directory on the storage device as the `Update` program itself and has to be called `Update.script`. Each line of the script contains one command to be executed. The command is identified by a single letter, followed by one or more parameters.

1. Installing a file: `I`
   This command installs a new file on the board. It might replace an existing file with a newer version.
2. Deleting a file: `D`
   This command removes a file from the board when it is no longer persistent in the new software version.
3. Executing a program: `X`
   This command executes a program that provides a part of the update process.
4. Setting a display pause: `P`
   This command sets the minimum time how long every update action remains visible on the display.
5. Setting a log file: `L`
   Sets the name of a file where the update actions are logged to be reviewed later.
6. Comments: `#`
   A comment line is completely ignored by the `Update` program. Empty lines in the script file are also ignored.

Using these commands you create a script file telling step by step how to do the update.

This is an example how an update script may look like.

```
# Sample Update Script
# This script updates a MyApp software to V2.7

# Open log file on update storage device
L Update-V2.7.log

# Run as fast as possible
P 0

# Delete some old log file
D \FFSDISK\MyApp.log

# Install the new application program
I MyApp-V2.7.exe \FFSDISK\MyApp.exe

# Install two data files
I MyData1-V2.7.bin \FFSDISK\MyData1.bin
I MyData2-V2.7.xyz \FFSDISK\MyData2.xyz
```

```
# Install a new help file
I MyApp-V2.7.hlp \FFSDISK\MayApp.hlp
```

3.3 Backup Files and Rollback

An important aspect of this `Update` program is the concept of backup files. As we've already said in the introduction, it is vital that the board is left in a working state after doing the update. So if the update fails anywhere, the previous state of the software should be restored so that at least the old software version can resume work.

This requires that we save the previous state somehow, and this is where backup files come into play. The idea is rather simple. Before installing a new file with `I` or deleting a file with `D`, the old version of the file is moved to a save position. Usually it is only renamed to make place for the new file, but you can explicitly give a path and file name where this safety copy, the so called backup file, will be stored. For example it can even be copied temporary to the update device.

When the whole update procedure succeeds without error, all these backup files are simply deleted. They are not required anymore. But if there happens some fatal error somewhere later in the update process, then all these backup files can be restored to their original place. This procedure is called "rollback. After rollback, the board is again in the previous state and the old software should work again.

This shows us how the `Update` program works:

1) Scan the script file to check for syntax errors. It's also verified that all new files really exist on the update device.
2) Do the update process, installing new files and deleting old files.
3)
    a) Update process succeeds: Remove all backup files
    b) Update process fails: Do rollback and restore old files

When doing rollback, it is important to bring the backup files back in the reverse order they were saved, because cyclic replacements of files can only be undone correctly in this way. And to keep the final step consistent, we also delete the backup files in reverse order in case of success.

3.4 File Names

When using file names in the update script, it is possible to use absolute or relative paths. An absolute path starts with character \ and refers to a path starting at the root directory, most probably the main flash memory located in \FFSDISK A relative path must not start with \ and refers to files relative to the current update directory on the update storage device.

You can also use / instead of \ as directory delimiter. If a path contains blanks, you have to enclose it in double quotes.

Examples:

Here and in all following examples we assume the update is done from an SD Card on a NetDCU8 resulting in the path "\Storage Card\NetDCU8". Then the following file name mapping takes place.

Given file name	*Final file name used*
\FFSDISK\MyFile	"\FFSDISK\MyFile"
\MyFile	"\MyFile"
MyFile	"\Storage Card\NetDCU8\MyFile"
"My File"	"\Storage Card\NetDCU8\My File"
"Storage Card\MyFile"	"\Storage Card\MyFile"

It is possible to give the empty file name "" as backup file name. But please note that the old file won't get a backup then and definitely can't be restored during rollback. If update fails, this file is irretrievably lost. So do this at your own risk!

3.5 Installing a file

Syntax:

```
I <source> <target> [<backup>]
```

Parameters:

`<source>`	File name of the source file (usually on the update storage device)
`<target>`	File name of the target file (usually in `\FFSDISK` on the NetDCU)
`<backup>`	File name of the backup file where the old target file is stored (optional). If not given, the default backup file name is `<target>.bak`, i.e. the target file name extended by `.bak`.

Description:

If source file `<source>` and target file `<target>` are identical (have the same content), no action is taken. Otherwise if the target file `<target>` exists, it is moved to the backup file name `<backup>`. Then in any case the source file `<source>` is copied to the target file name `<target>`.

Example 1:

```
I  New.exe  \FFSDISK\CurrentApplication.exe
```

Install the new file `"\Storage Card\NetDCU8\New.exe"` as `"\FFSDISK\CurrentApplication.exe"` on the board. If this file exists, it is stored for backup under the name `"\FFSDISK\CurrentApplication.exe.bak"`.

Example 2:

```
I  NewData  \FFSDISK\Data  OldData.mybak
```

Install the new file `"\Storage Card\NetDCU8\NewData"` as `"\FFSDISK\Data"` on the NetDCU. If this file exists, it is copied to the update storage device for backup under the name `"\Storage Card\NetDCU8\OldData.mybak"`.

Action during rollback:

The current target file `<target>` is deleted and the backup file `<backup>` is moved back to the target file name `<target>`.

3.6 Deleting a file

Syntax:

`D <target> [<backup>]`

Parameters:

`<target>`	File name of the target file (usually in `\FFSDISK` on the NetDCU)
`<backup>`	File name of the backup file where the old target file is stored (optional). If not given, the default backup file name is `<target>.bak`, i.e. the target file name extended by `.bak`.

Description:

If the target file `<target>` exists, it is moved to the backup file name `<backup>`.

Example:

```
D  \FFSDISK\Logfile.txt
```

Delete the file `"\FFSDISK\Logfile.txt"` while using the name `"\FFSDISK\Logfile.txt.bak"` as backup name.

Action during rollback:

The backup file `<backup>` is moved back to the target file name `<target>`.

3.7 Executing a program

Syntax:

```
X '<dcommand>' ['<ucommand>']
```

Parameters:

`<dcommand>` Command line for the `DO` program, enclosed in single quotes. This program does the update part.

`<ucommand>` Command line for the `UNDO` program, enclosed in single quotes. This program does the rollback part in case of an update failure.

Description:

During installation, the `DO` program is called with the `dcommand` command line. If the program returns zero, it is assumed to have succeeded and the installation process continues with the next script line. If the program returns a non-zero value, it is supposed to have failed, and the rollback procedure is started.

The second command line, the `UNDO` program `ucommand`, is only executed, when the installation fails at some point (now or later) and the rollback process comes alive. This program should undo everything that was done in the `DO` program.

Even as the command lines are enclosed in single quotes `'...'`, the program name (the first argument in each command line) is expanded as in every other file, i.e. the path of the update device is prepended if the file does not start with `\` or `/` (see page 15). In addition the program extension (`.exe`, `.bat`, etc.) is automatically appended if not specified.

But please note that the `Update` program does not know whether other arguments in the command line string are file names or not. Therefore it cannot expand other arguments in the same way. Either this has to be done in the called program, or the files have to be given with the absolute paths already in the script.

Using single quotes to enclose the command line allows the usage of double quotes inside the arguments themselves, e.g. to enclose an argument that includes a space character.

Example:

```
X 'writeboot /e:eboot8_117.nb0 /a' 'writeboot /e:eboot8_116.nb0 /a'
```

Call program `writeboot.exe` located on the update device to update the boot loader to the newer version V117. If the installation fails at some point, call `writeboot.exe` again and restore the older boot loader V116.

Action during rollback:

The `UNDO` program is called with the `ucommand` command line. This program should undo everything that the `DO` program did during install. A return value of zero reports success, a non-zero value reports failure.

If no `UNDO` program is given, the rollback cannot re-establish the previous state of this part of the installation.

3.8 Setting the display pause

Syntax:

`P <time>`

Parameters:

`<time>` Minimum time to show update actions (in ms)

Description:

Every action of the update process is shown in a dialog box on the display. By default, every command is shown at least one second. If you give this command, all further actions will be shown for at least the new time.

You can give this command in the script file as often as you like. For example to delay important commands longer than unimportant ones.

Please note that every command can take more than one action to complete. For example a file installation with `I` can take up to six separate actions that are shown separately: compare source and target, copy target to backup, rename backup, delete target, copy source to target, rename target.

Examples:

```
P 0
```

This processes the update script as fast as possible, not waiting for output at all.

```
P 4500
```

Wait at least 4.5 seconds before showing the next action.

3.9 Setting a log file

Syntax:

`L <logfile>`

Parameters:

`<logfile>` Log file name where update actions are stored

Description:

The steps of the update process can also be logged to a file to be reviewed later. Every action is stored as a couple of lines showing the type of action, the script line number, the "from" and "to" file names and the result of the operation.

As already mentioned, a command may be subdivided into several actions. Two actions are separated in the log file by a blank line, actions of different command lines are separated by a string of dashes "`-----`".

Usually you'll only give one log file command at the beginning of the script file. Every new `L` command found later in the script closes the current log file and opens the new one.

Please note that the syntax check of the script file and the existence check of the source files is not logged. Logging starts when the main update process begins. So errors in this first step of the update can only be seen on the display.

Every start of the update procedure will start the log file anew. Please keep this in mind when you restart an interrupted update process or you'll overwrite the log file of the first part of the update.

Example:

```
L Update.log
```

Use "`\Storage Card\NetDCU8\Update.log`" for storing the log file.

The resulting log file may look like this:

```
# Log file created by NetDCU Update V1.0

Update: Checking file
Script line: 6
File: \FFSDISK\Logfile.txt
Result: Different, doing update

Update: Renaming file
Script line: 6
From: \FFSDISK\Logfile.txt
To: \FFSDISK\Logfile.txt.bak
Result: OK
```

```
-----

Update: Comparing source and target
Script line: 7
From: \Hard Disk\NetDCU8\NewApplication.exe
To: \FFSDISK\CurrentApplication.exe
Result: Different, doing update

Update: Renaming file
Script line: 7
From: \FFSDISK\CurrentApplication.exe
To: \FFSDISK\CurrentApplication.exe.bak
Result: OK
Update: Copying file
Script line: 7
From: \Hard Disk\NetDCU8\NewApplication.exe
To: \FFSDISK\CurrentApplication.exe.tmp
Result: OK

Update: Renaming file
Script line: 7
From: \FFSDISK\CurrentApplication.exe.tmp
To: \FFSDISK\CurrentApplication.exe
Result: OK

-----

Update: Comparing source and target
Script line: 8
From: \Hard Disk\NetDCU8\NewData.bin
To: \FFSDISK\Data.bin
Result: Different, doing update

Update: Renaming file
Script line: 8
From: \FFSDISK\Data.bin
```

```
To: \FFSDISK\OldData.mybak
Result: OK

Update: Copying file
Script line: 8
From: \Hard Disk\NetDCU8\NewData.bin
To: \FFSDISK\Data.bin.tmp
Result: OK

Update: Renaming file
Script line: 8
From: \FFSDISK\Data.bin.tmp
To: \FFSDISK\Data.bin
Result: OK

-----

Update: Deleting backup file
Script line: 8
File: \FFSDISK\OldData.mybak
Result: OK

Update: Deleting backup file
Script line: 7
File: \FFSDISK\CurrentApplication.exe.bak
Result: OK

Update: Deleting backup file
Script line: 6
File: \FFSDISK\Logfile.txt.bak
Result: OK
```

3.10 Repeating the Update Process

There is no danger in repeating the update procedure on a NetDCU board that was already updated previously. Before a file is replaced, it is compared to the new file on the update storage device. If they are identical, having exactly the same content, the file is assumed to be up-to-date and no new update takes place.

So if an already updated NetDCU is updated again, the log file will simply show that all files were up-to-date.

3.11 Resuming Interrupted Updates

If you have inserted the update storage device inadvertently into the NetDCU, it immediately activates the `BootDelay`. But don't be afraid, you can safely pull out the storage device and nothing happens. On the next reboot, the boot delay stage will take place, but as the update storage device is not present anymore, no `Update` program is found and the `BootDelay` will simply count down and time out. Then the main application will start again and the board will behave as if nothing had happened.

So it is not necessary to interrupt the update process by a forced removing of the update storage device during update. However the concept of this `Update` mechanism is designed in a way that it also allows to resume an interrupted update, for example after an unexpected power failure while updating.

To achieve this goal, it is important to be able to decide if a file copy was successfully completed or only done in part. This is achieved by copying to temporary file names.

Copying a file is a rather time consuming action. It may take several seconds or even minutes, depending on the size of the file. Renaming a file on the other hand is a rather short operation that can be seen as an atomic file operation.

So instead of directly copying a file to its target name, we first copy it to a temporary file on the target device. The temporary file name is built by appending `.tmp` to the target file name. If the copy succeeds, we can immediately rename the file to its final target name and are done.

The advantage of this procedure is, that when we later resume such an interrupted copy procedure, we can immediately say if the copy was complete or not. If the target file exists, the copy was complete. If the target file does not exist and instead only the temporary file exists, the copying was incomplete and has to be resumed or redone.

Example:

```
I NewData.bin \FFSDISK\Data.bin
```

After having created the backup file, the final copy process uses these steps:

1. Copy "`\Storage Card\NetDCU8\NewData.bin`" to "`\FFSDISK\Data.bin.tmp`"
2. Rename temporary file "`\FFSDISK\Data.bin.tmp`" to the final target file "`\FFSDISK\Data.bin`".

These two actions can be seen as separate steps in the dialog box on the display or in the log file. The same copy procedure is used when creating the backup file and the backup file is on a different device than the target file.

So when doing the update, the actions taken depend on the situation found when starting the command. We consider at least the following cases:

Install `I`:

1. Target exists, source and target are identical
   Update already succeeded, do nothing.
2. Target exists, there is no backup
   Move target to backup, copy source to target.

3. Target exists, backup exists
   Update was interrupted earlier after having copied the file to the backup but before deleting the target. Resume by deleting target and then copy source to target.
4. Target does not exist, there is no backup
   The file did not exist before. Copy source to target.
5. Target does not exist, backup exists
   Update was interrupted earlier after having moved the target to the backup. Resume by copying source to target.

Delete `D`:

1. Target exists, there is no backup
   Move target to backup
2. Target exists, backup exists
   Update was interrupted earlier after having copied the file to the backup but before deleting the target. Just delete the target.
3. Target does not exist, there is no backup
   This file never existed, so don't do anything.
4. Target does not exist, backup exists
   Update was interrupted earlier after having moved the file to the backup. So this step is already complete, resume by doing nothing.

Execute `X`:

There is no way to tell if the program already succeeded earlier or not. The program will be called in any case. Therefore the program itself has to decide if the update step still needs to take place or is already done.

During rollback, the action also depends on the situation:

Install `I`:

1. Target exists, backup exists
   No error in this file. Delete target, move backup to target.
2. Target does not exist, backup exists
   The error happened after having moved the target to the backup. Resume by moving backup back to target.
3. Target does not exist, backup does not exist.
   Target never existed, the error happened while copying the source to the target. Nothing to do.

Delete `D`:

1. Target exists, backup exists
   The error happened after having copied the file to the backup but before deleting the target. Resume by deleting target, then move backup to target.
2. Target does not exist, backup exists
   No error in this file. Move backup to target.
3. Target does not exist, backup does not exist
   This file never existed. Nothing to do.

In fact, the real analysis of the situation as implemented in the `Update` program is still more complicated!

Execute x:

The `UNDO` program will be called in any case. Therefore the program itself has to decide how to do the rollback.

Please note:

Even if this whole update concept allows resuming after interrupts, this is only meant as an emergency recovery from an accidental maloperation or unexpected power loss. It is not meant as a normal way of doing the update in steps. We can never guarantee that interrupted file operations will keep a working and consistent file system. The file system may be damaged to a point beyond repair in such a case, so that only complete erasure of the flash memory and a new download of the Windows CE kernel can revive the board. So please avoid interrupting the `Update` program whenever possible.

But when the update *was* interrupted, it has to be resumed after the next start of the NetDCU board. Because of this the `Update` program keeps the `BootDelay` value in the registry active until the whole update procedure is completely done. Then the value is reset to 0.

3.12 Summary

Let's summarise the steps for doing a software update on a NetDCU using the mechanism presented in this document.

These steps are required that your NetDCU board is capable of doing an automatic update:

- Verify that `CheckAutoStart.exe` is started in `init`.
- Make sure that your application depends on `CheckAutoStart.exe` when starting.

Now the steps to prepare a storage device for update. We assume a NetDCU8 here.

- Create a directory `NetDCU8` on your update storage device and put the `Update` program there under the name `AutoStart.exe`.
- Write an update script with all the commands to execute when doing the update. Put it as `Update.script` in the `NetDCU8` directory on your update storage device.
- Put all the required new files of your application in the `NetDCU8` directory on your update storage device. You can use subdirectories as you like.

These steps are required to perform the actual update:

- Plug the update storage device in the running NetDCU. Wait until requested to reboot the board.
- Shut down the main application and reboot the board. The update takes place now.
- When requested after the update is done, remove the update storage device and reboot the board.

4 Appendix

Important Notice

The information in this publication has been carefully checked and is believed to be entirely accurate at the time of publication. F&S Elektronik Systeme assumes no responsibility, however, for possible errors or omissions, or for any consequences resulting from the use of the information contained in this documentation.

F&S Elektronik Systeme reserves the right to make changes in its products or product specifications or product documentation with the intent to improve function or design at any time and without notice and is not required to update this documentation to reflect such changes.

F&S Elektronik Systeme makes no warranty or guarantee regarding the suitability of its products for any particular purpose, nor does F&S Elektronik Systeme assume any liability arising out of the documentation or use of any product and specifically disclaims any and all liability, including without limitation any consequential or incidental damages.

Specific testing of all parameters of each device is not necessarily performed unless required by law or regulation.

Products are not designed, intended, or authorized for use as components in systems intended for applications intended to support or sustain life, or for any other application in which the failure of the product from F&S Elektronik Systeme could create a situation where personal injury or death may occur. Should the Buyer purchase or use a F&S Elektronik Systeme product for any such unintended or unauthorized application, the Buyer shall indemnify and hold F&S Elektronik Systeme and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, expenses, and reasonable attorney fees arising out of, either directly or indirectly, any claim of personal injury or death that may be associated with such unintended or unauthorized use, even if such claim alleges that F&S Elektronik Systeme was negligent regarding the design or manufacture of said product.

Specifications are subject to change without notice.

Warranty Terms

Hardware Warranties

F&S guarantees hardware products against defects in workmanship and material for a period of two (2) years from the date of shipment. Your sole remedy and F&S's sole liability shall be for F&S, at its sole discretion, to either repair or replace the defective hardware product at no charge or to refund the purchase price. Shipment costs in both directions are the responsibility of the customer. This warranty is void if the hardware product has been altered or damaged by accident, misuse or abuse.

Software Warranties

Software is provided "AS IS". F&S makes no warranties, either express or implied, with regard to the software object code or software source code either or with respect to any third party materials or intellectual property obtained from third parties. F&S makes no warranty that the software is useable or fit for any particular purpose. This warranty replaces all other warranties written or unwritten. F&S expressly disclaims any such warranties. In no case shall F&S be liable for any consequential damages.

Disclaimer of Warranty

THIS WARRANTY IS MADE IN PLACE OF ANY OTHER WARRANTY, WHETHER EXPRESSED, OR IMPLIED, OF MERCHANTABILITY, FITNESS FOR A SPECIFIC PURPOSE, NON-INFRINGEMENT OR THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION, EXCEPT THE WARRANTY EXPRESSLY STATED HEREIN. THE REMEDIES SET FORTH HEREIN SHALL BE THE SOLE AND EXCLUSIVE REMEDIES OF ANY PURCHASER WITH RESPECT TO ANY DEFECTIVE PRODUCT.

Limitation on Liability

UNDER NO CIRCUMSTANCES SHALL F&S BE LIABLE FOR ANY LOSS, DAMAGE OR EXPENSE SUFFERED OR INCURRED WITH RESPECT TO ANY DEFECTIVE PRODUCT. IN NO EVENT SHALL F&S BE LIABLE FOR ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES THAT YOU MAY SUFFER DIRECTLY OR INDIRECTLY FROM USE OF ANY PRODUCT. BY ORDERING THE PRODUCT, THE CUSTOMER APPROVES THAT THE F&S PRODUCT, HARDWARE AND SOFTWARE, WAS THOROUGHLY TESTED AND HAS MET THE CUSTOMER'S REQUIREMETS AND SPECIFICATIONS