Kernel UpDate Tool

UpDate kernel image during runtime
Windows Embedded Compact

Version 1.13
2015-10-06

Elektronik Systeme

Untere Waldplätze 23
D-70569 Stuttgart
Fon: +49(0)711-123722-0
Fax: +49(0)711 – 123722-99

History

Date	V	Platform	A,M,R	Chapter	Description	Au
2007-09-01	1.0	-	A	*	First revision	MK
2007-11-10	1.1	PicoMOD1	A	1.2	Support for PicoMOD1 added.	MK
2009-06-02	1.2	NetDCU11	A	1.2	Support for NetDCU11 added. A4 document layout.	MK
2009-06-02	1.3	PicoMOD3	A	1.2	Support for PicoMOD3 added.	MK
2009-06-23	1.4	PicoCOM1/2	A	1.2	Support for PicoCOM1/2 added.	MK
2009-12-15	1.5	PicoMOD4	A	1.2	Support for PicoMOD4 added.	MK
2010-02-04	1.6	*	A	3	Kernel Update library documentation added.	MK
2011-09-14	1.7	*	A	*	XIP Kernel image support documented.	MK
2011-09-21	1.8	*	A	2.3	Option to disable up-to-date check added.	MK
2012-01-08	1.9	*	A,M	*	General documentation update. Support for FSS5PV201 added.	MK
2014-01-21	1.10	*	A, M	*	General documentation update. Support for FSVybrid and FSIMX6 added.	ZU
2015-05-18	1.11		M	*	Minor layout corrections	HF
2015-08-05	1.12		A	About	Added new remark regarding name of application.	HF
2015-10-06	1.13		M	2.3	Option to finish update successfully if up-to-date check is true.	ZU

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

About This Document

This document describes how to use the Kernel Update program, which is part of the NetDCU-SKIT-USB-UpDate software package. A list of all platforms this utility is available for can be found in chapter 1.2.

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 KernelUpdate.exe (Windows Embedded CE 6.0 and Windows Embedded Compact 7) or KernelUpdate2013.exe for Windows Embedded Compact 2013.
In the remaining document we'll use the term "KernelUpdate" 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 Content

1 Introduction

To update the currently used kernel on NetDCU, programs like USB-Loader or Eshell are used, generally. This is a common method during development. On “finalized” systems this often isn’t possible, as the needed interfaces are customized or not accessible anymore.

The main goal of this update program is to enable kernel updates, during the operating system is running, very easily. In combination with the software update tool, this can even be managed automatically.

1.1 Update process

First of all please keep in mind that **updating the kernel always is a dangerous action** and could result in an unbootable system. It’s always advisable to test a kernel update with one device, before updating a wide range of devices.

Kernel functionality grows during time. For this reason there are some (old) kernels that don’t include the requirement for a proper working kernel update.

To keep the risk of damaging the board as small as possible, executing all needed tasks is ordered in a specific way and `KernelUpdate` will try to intercept most possible situations. This of course includes some basic verification. For example it will check if the current size of kernel partition is big enough to hold the kernel or if the given binary file really contains the expected kernel signature. Additionally the bin file will be written in a “testmode” and can be aborted before performing real writes.

1.1.1 Kernel Update Checklist

To make sure that kernel updating can be executed properly there are some tips you should take into account:

- ✓ Close user application(s) during kernel update. You might take advantage of the update program from F&S to arrange this. **Please refer to the “CheckAutoStart and Update” documentation for details.**
- ✓ **Please do not start any programs during update process.**
- ✓ **KernelUpdate** program and **kernel image** to be flashed may ***not* be located on NAND flash** (FFSDISK).
- ✓ Verify the update process on a testing system before applying to a wide range of devices. And make sure this test has been arranged under same conditions as in the field.
- ✓ Do not remove external devices during update. Especially removing the device, the kernel image is located on may definitely damage your system.

1.2 Supported Platforms

Following table shows all platforms the program `KernelUpdate.exe` is available for. Please notice that the `KernelUpdate.exe` program requires support for flash access within the kernel. For that reason the kernel version this support has been added into kernel is also listed below.

Platform	Kernel version requirements
NetDCU	
NetDCU5.2	---
NetDCU8	---
NetDCU9	V1.15
NetDCU10	---
NetDCU11	V1.15 (090528)
NetDCUA5 FSVybrid	V1.09
PicoMOD	
PicoMOD1	---
PicoMOD3	V0.09 (080718)
PicoMOD4	-
PicoMOD6	V1.09 (110914)
PicoMOD7A FSS5PV210	V1.12 (121219)
PicoMODA9 FSIMX6	V1.00
PicoMOD1.2 FSVybrid	V1.00
PicoMODA5 FSVybrid	V1.00
PicoCOM	
PicoCOM1	V1.11 (090617)
PicoCOM2	V1.04 (090623)
PicoCOM3	V1.09 (110914)
PicoCOM4	V1.09 (110914)
PicoCOMA5 FSVybrid	V1.09

Platform	Kernel version requirements
QBliss	
QBlissA8	V1.08 (110601)
QBlissA9 FSIMX6	V1.00
armStone	
armStoneA5 FSVybrid	V1.09
armStoneA8 FSS5PV210	V1.12 (121219)
armStoneA9 FSIMX6	V1.00
nanoRISC	
nanoRISCA8	V1.12 (121219)
efus	
efusA9 FSIMX6	V1.0

Table 1: Supported platforms and their kernel version requirements.

2 The Kernel Update Program

The `KernelUpdate` program includes a dialog that will display all steps and their results. If this wouldn't be desired there's a "quiet mode" available, to disable this dialog.

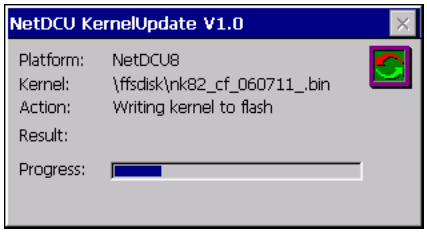

Figure 1: Screenshot during kernel write

Optionally it's possible to force `KernelUpdate` to log all performed actions and results to a logfile:

Example:

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

Action: Find kernel bin file
Kernel: \ffsdisk\nk82_cf_060711.bin
Result: OK

Action: Checking Platform
Platform: NetDCU8
Result: OK

Action: Checking kernel signature
Result: OK

Action: Checking size of kernel partition
Result: OK

Action: Writing kernel in "Testmode"
Result: OK

Action: Writing kernel to flash
Result: OK

Result: Done

Update completed successfully
```

Listing 1: Log file example.

2.1 XIP Kernel (experimental)

On latest platforms (since PicoMOD6) we support a special kernel image variant, called XIP (eXecute In Place) kernel. Instead of a "regular" kernel image, this image must not be held in system memory completely. System components and modules will be loaded from NAND flash just as they are needed. Using this variant features some fundamental advantages like faster bootup times and less system memory usage. But related to runtime kernel image updating it challenges much higher software requirement. It might be possible that the system tries to load some data from kernel partition during or after updating process, which might lead into system instability**. Therefore XIP kernel image support is declared to be experimental in current release.**

To avoid the risk of data loss, kernel update program will verify that all required modules are loaded completely and dismount kernel partition. But because of this fact, operating system may no longer function properly after kernel update has finished. **Hence the system will reboot automatically after kernel update has been finished successfully.**

This should be catched up as a workaround solution to update XIP kernel images yet. F&S already is working on a more flexible kernel update procedure.

2.2 Compressed Kernel

On NetDCU52 and NetDCU6 there's an option to store the kernel compressed, which helps to reduce flash memory consumption. This option can't be changed with `KernelUpdate`.

If compression is enabled in bootloader, the structure of WindowsCE binary image format (`.bin`) causes principle problems and can't be used by `KernelUpdate`. The image therefore has to be available in raw format (`.nb0`). This can be achieved with the also attached program `cvrtbin`, which is attached to this software package.

Following example shows how to convert an available `bin` file to the needed raw `nb0` file.

Example:

```
C:\temp>cvrtbin NK52C1_070301.bin
ViewBin... NK52C1_070301.bin
Image Start = 0x80120000, length = 0x00BD4658
                Start address = 0x80121000
Checking record #93 for potential TOC (ROMOFFSET = 0x00000000)
Found pTOC  = 0x80cf2d54
ROMOFFSET = 0x00000000
Done.

C:\temp>cvrtbin -r -a 0x80120000 -l 0x00BD4658 -w 32 NK52C1_070301.bin
ViewBin... NK52C1_070301.bin
Image Start = 0x80120000, length = 0x00BD4658
                Start address = 0x80121000
Checking record #93 for potential TOC (ROMOFFSET = 0x00000000)
Found pTOC  = 0x80cf2d54
ROMOFFSET = 0x00000000
start 80120000 length 00000004
start 80120040 length 00000008
.
.
.
start 80cf2da8 length 000018b0
Progress...
0%Done.
```

Listing 2: Converting a bin file. Only required when updating compressed kernel images.

After executing the last command, a `.nb0` file with the same name should be available (`NK52C1_070301.bin` in this case).

2.3 Execution

Syntax:

```
KernelUpdate -k <kernelfile> [-l <logfile>]
             [-q] [-s <start-address>]
             [-a <launch-address>]
             [-r [<timeout>]]
             [-f [<up to date action>]]
```

Parameters:

`-k <kernelfile>`
Path to the kernel bin file to be written.

`-l <logfile>`
Path of the logfile.

`-q` This option causes `KernelUpdate` to suppress displaying the dialog.

`-r [<timeout>]`
Reboot system automatically if kernel has been updated successfully. If an error has occurred this option is discarded.
This option is used automatically if a XIP image is updated (see 2.1).

`-f [<up to date action>]`
Without "-f" an update is abort if the kernel is already up to date
Parameter values:

0 (default)	force update and skip up to date check.
1	skip update and finish update successfully if kernel is up to date.
Else	ignored

`-s <start-address>`
Store the given start-address in bootloader. Only available when writing compressed images.

`-a <launch-address>`
Store the given launch-address in bootloader. Only available when writing compressed images.

`-u`
Skip unmounting BINFS which is default when updating XIP kernel images.

Return values:

- `0` Update finished successfully
- `1` Update aborted. The kernel has NOT been updated and NetDCU is in state as it was before.

2 Update failed. An error occurred during update that could have damaged NetDCU. It's not guaranteed that reboot will be able.

3 Kernel Update library

The KernelUdpate library is designed to integrate the functionality of the common KernelUpdate program into your own application. Both possibilities are built from the same code base and therefore should work nearly identical.

The process of updating a kernel image separates into four stages.

1. **Initialization:**
   By initializing the kernel update library there will already evaluated some simple requirements. Additionally the application gets detailed information about the next steps to be performed during update.
2. **Checking requirements:**
   Before writing the kernel into flash it is mandatory to check some requirements and make sure that updating can be executed without any problems on the current system.
3. **Writing the Kernel into flash:**
   After all requirements are validated the kernel can be written to flash permanently. To enhance safety, the KernelUpdate library also supports simulating the flash writes.
4. **Query update result and deinitialize KernelUpdate library:**
   After the kernel has been flashed the application might want to get detailed information about the updating result. Additionally the KernelUpdate library must be deinitialized to free all temporarily allocated resources.
5. **Reboot device**:
   When using KernelUpdate library the board will not be rebooted automatically. But especially when updating a XIP kernel image it is *highly recommended* to reboot the device quickly as the kernel partition is no longer valid or even available (compare 2.1).

Note:
The Kernel Update library is only available on request. It is not included in the CD you have received with SKIT-UPDATE package. Please contact our support team to get more information.

3.1 Example

```
#include <windows.h>
#include "KernelUpdateLib.h"

/* … * /

TCHAR szKernelFile[MAX_PATH];
UINT uError = ERROR_SUCCESS;
KERNEL_UPDATE_PARMS cKernelUpdateParms;
DWORD dwNumReqs;
TCHAR szStatusText[MAX_STATUS_TEST];
CString strStatusMessage;
DWORD dwProgress;


// Get the location of the kernel bin file
m_objCEditKernelFile.GetWindowTextW(szKernelFile, MAX_PATH);



// Stage 1 - Initialize kernel update library
wcsncpy(cKernelUpdateParms.szKernelFile, szKernelFile, MAX_PATH);
cKernelUpdateParms.bVerbose = FALSE;  // Disable verbosity
cKernelUpdateParms.bAutoCheck = TRUE; // Check all requirements
            //automatically.
cKernelUpdateParms.bForceUpdate = FALSE; // Force Update even
                                         // when the same kernel is
                                         // running already
uError = InitKernelUpdate(&cKernelUpdateParms, &dwNumReqs);



// Stage 2 - Verify update requirements
// NOTE: As we have enable auto-checking, this will already be performed
// by InitKernelUpdate()
#if 0
if (ERROR_SUCCESS == uError)
{
    DWORD dwCurStep = 1;
    do
    {
        GetRequirementDescription(szStatusText, MAX_STATUS_TEST,
                                  dwCurStep);
        uError = CheckNextRequirement(dwCurStep++);
    } while (ERROR_SUCCESS == uError);
    if ((ERROR_SUCCESS != uError) && (KU_ERROR_NOMORE_REQS != uError))
    {
        strStatusMessage.Format(L"%s - FAILED", szStatusText);
        MessageBox(strStatusMessage);
        return;
    }
}
else
#else
if (ERROR_SUCCESS != uError)
#endif
{
    strStatusMessage.Format(L"Failed to initialize KernelUpdate \
                                          library (error: %d)",
                            uError);
    MessageBox(strStatusMessage);
    return;
}

// Stage 3- Start kernel update
// Simulate first
uError = StartKernelUpdate(TRUE);
if (ERROR_SUCCESS != uError)
{
    strStatusMessage.Format(L"Could not start Kernel update \
                                    (simluation mode)(error:%d)",
                            uError);
    MessageBox(strStatusMessage);
    return;
}
else
{
```

```
    m_objCProgressProgress.SetRange32(0, 100);
    while(GetCurrentUpdateProgress(&dwProgress))
    {
        // update progress bar
        m_objCProgressProgress.SetPos(dwProgress);
    }
}
// Query result
uError = GetUpdateResult();
if (ERROR_SUCCESS != uError)
        strStatusMessage.Format(L"Updating failed!! (error:%d)",
                                uError);
m_objCProgressProgress.SetPos(0);

// Write kernel to flash pernanetly
uError = StartKernelUpdate(FALSE);
if (ERROR_SUCCESS != uError)
{
    strStatusMessage.Format(L"Could not start Kernel update\
                                                    (error:%d)",
                            uError);
    MessageBox(strStatusMessage);
    return;
}
else
{
    m_objCProgressProgress.SetRange32(0, 100);
    while(GetCurrentUpdateProgress(&dwProgress))
    {
        // update progress bar
        m_objCProgressProgress.SetPos(dwProgress);
    }
}


// Stage 4 - Query update result
uError = GetUpdateResult();
if (ERROR_SUCCESS != uError)
    strStatusMessage.Format(L"Updating failed!! (error:%d)",
                            uError);
else
    strStatusMessage.Format(L"Kernel updated successfully!!");
MessageBox(strStatusMessage);

return;
```

3.2 How to interagrate Kernel Update library into your application

The Kernel Update library is named `KernelUpdateLibrary.lib` . To integrate the library into your application you simple have to include the header file (`KernelUpdateLib.h`) and tell the linker where the implementation can be found.

Figure 2 shows a screenshot of the require configuration within your Visual Studio Project.

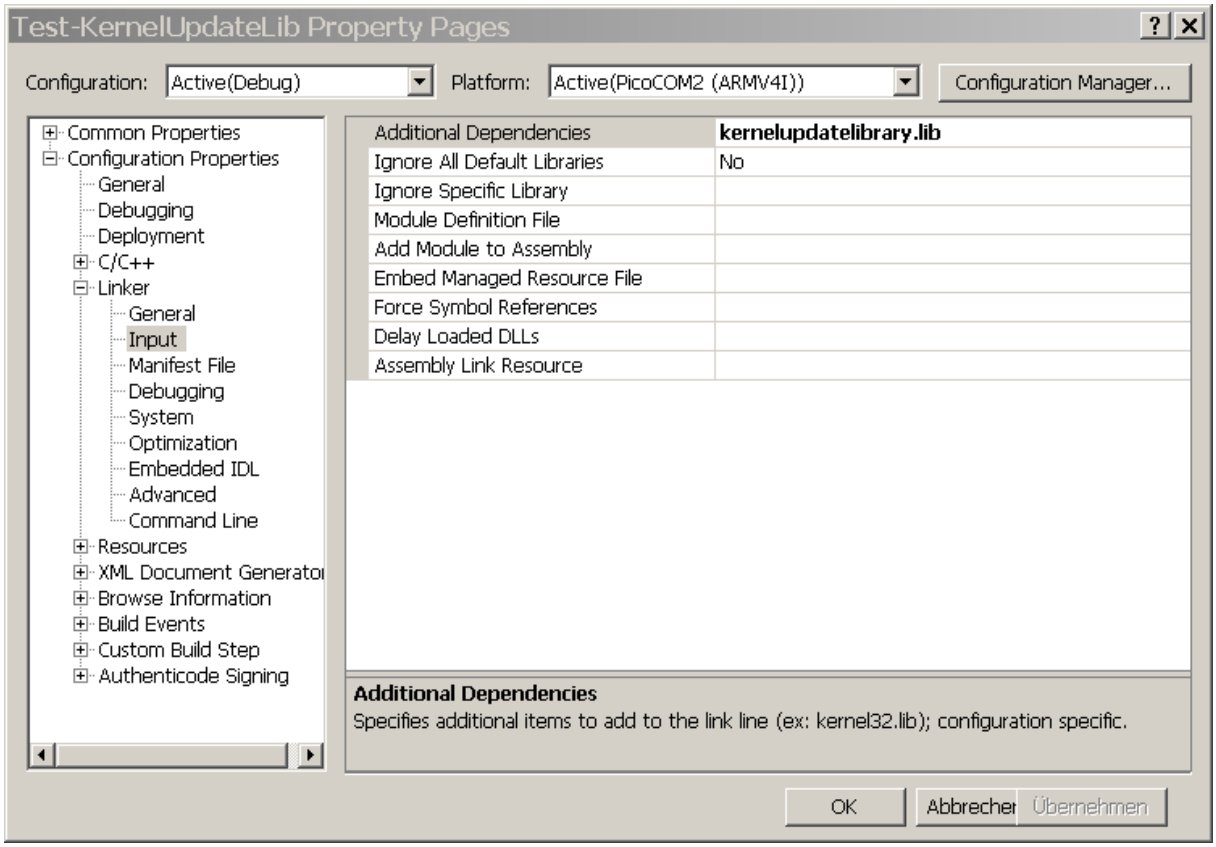

Figure 2: Visual Studio settings to integrate Kernel Update library.

3.3 API reference

3.3.1 InitKernelUpdate()

Signature:

```
UINT InitKernelUpdate(PKERNEL_UPDATE_PARMS pParms,
                      PDWORD dwNumReqs);
```

Parameters:

`pParms`	Pointer to a `KERNEL_UPDATE_PARMS` structure, defining some configuration values for Kernel Update.
`dwNumReqs`	(optional) The number of available requirements to be checked will be set in this variable.

`KERNEL_UPDATE_PARMS` structure:

`bVerbose`	Enable / disable verbosity.
`szKernelFile`	Location of the kernel file.
`bAutoCheck`	Start requirement check automatically.

Return:

`KU_ERROR_SUCCESS`	Success
Otherwise	Error value (see 3.3.7)

Description:

This is the first function that must be called to initialize the Kernel Update library. Additionally the application will receive information about the number of dependencies.

3.3.2 CheckNextRequirement()

Signature:

```
UINT CheckNextRequirement(UINT uStep=1)
```

Parameters:

`uStep`	(optional) Current dependency check that should be performed.

Return:

`KU_ERROR_NO_MORE_REQS`	All requirements have been checked.
`KU_ERROR_SUCCESS`	As long as there are more steps to be performed the function will return `ERROR_SUCCESS`.
Otherwise	Error value (see 3.3.7)

Description:

After initializing the Kernel Update library some simple requirements have already been checked. But to guarantee that updating the kernel can be performed successfully; there are some more issues to be checked. The overall number of requirements for your platform are already defined by `InitializekernelUpdate()`.

Each time this function is being called, the next dependency will be checked automatically. Therefore it is not required to defined the current step (`uStep`). If all requirements have been verified the function returns `KU_ERROR_NO_MORE_DEPS`. The function will return `ERROR_SUCCESS` as long as there are more steps to be performed. Each other return code than `ERROR_SUCCESS` indicates an error. See 3.3.7 for details about the return error code.

3.3.3 StartKernelUpdate()

Signature:

```
UINT StartKernelUpdate(BOOL bSimulate, DWORD dwDelayMs=0)
```

Parameters:

`bSimulate`	If `TRUE` the kernel will be written in simulation-mode.
`dwDelayMs`	(optional) You can define a delay to wait before start writing to flash.

Return:

`KU_ERROR_SUCCESS`	Update thread started successfully.
Otherwise	Error value (see 3.3.7)

Description:

Not before checking all requirements the kernel can be written to flash. Additionally this function features the possibility to enable a simulation-mode. This helps to extend safety of updating process. The kernel file will be analysed in more detail, but no data will be written to flash.

3.3.4 GetCurrentUpdateProgress()

Signature:

```
BOOL GetCurrentUpdateProgress(PDWORD pdwCurProgress)
```

Parameters:

`pdwCurProgress`	The current progress (in percent) will be set in this variable. (0 ≤ `(*pdwCurProgress)` ≤ 100)

Return:

`TRUE`	Writing still in progress.
`FALSE`	Writing thread has finished. See `GetUpdateResult()` to get information about success.

Description:

In most cases the application may display the current state of updating using a progress bar. Therefore this function can be used after starting the update process (`StartKernelUpdat()`). This function will be blocking as long as the internal write thread of kernel library notifies about an progress update. This function will return as long as there are more portions to be written to flash. If the function returns `TRUE` this can either mean that updating has finished or there occurred an error during writing. To get detailed result information please call `GetUpdateResult()`.

3.3.5 GetUpdateResult()

Signature:

```
UINT GetUpdateResult(VOID)
```

Parameters:

–	-

Return:

KU_ERROR_SUCCESS	The Kernel has been written to flash successfully.
Otherwise	Error during update. See 3.3.7 for detailed error information.

Description:

As writing the kernel to flash will be performed in a custom thread, the overall result of updating must be queried using this function. If the kernel has been written successfully the return value will be KU_ERROR_SUCCESS. A detailed list of all possible return values can be found at the following table.

3.3.6 DeinitKernelUpdate()

Signature:

```
UINT DeinitKernelUpdate(VOID)
```

Parameters:

–	-

Return:

KU_ERROR_SUCCESS	The Kernel Update library has been deinitialized correctly.
Otherwise	See 3.3.7 for detailed error information.

Description:

After updating is finished all internal allocated resources must be freed again. To do so this function be called. Please note calling this function while updating is in process will abort all current routines and might lead into a damaged system.

3.3.7 Return codes

Return code	Description
KU_ERROR_SUCCESS ERROR_SUCCESS	Operation has been executed with no error.
KU_ERROR_FILENAME	Illegal file name.
KU_ERROR_PROGRAMM	Non-zero return value. Call `GetLastError()` to get information about the system error.
KU_ERROR_READ	Read-error
KU_ERROR_KERNEL_NOT_FOUND	Kernel file not found.
KU_ERROR_KERNEL_LOC_NOT_FOUND	Location of kernel file not found.
KU_ERROR_INVALID_KERNSIG	Invalid Kernel signature. The given file is not kernel image.
KU_ERROR_PLATFORM	Platform not supported.
KU_ERROR_WRITE	Write fault
KU_ERROR_CHECKSUM	Checksum Error. The kernel bin file seems to be corrupted.
KU_ERROR_WRITEPSD	Writing bootloader configuration failed.
KU_ERROR_OUTOFMEM	Of of memory.
KU_ERROR_KERNPART	Kernel partition to small.
KU_ERROR_READPSD	Reading bootloader configuration failed.
KU_ERROR_ABORTED	Update has been aborted.
KU_ERROR_NOT_READY	Update process not ready. This value will be returned if a functions is called before the internal state machine is ready for this action.
KU_ERROR_NOMORE_REQS	No more requirement validations available.
KU_ERROR_COMPRESSION_ENABLED	- *obsolete* - Compression activated in bootloder. The kernel can not be updated.
KU_ERROR_NO_COMPRESSION	- *obsolete* - Compression not activated
KU_ERROR_KERNSTART	- *obsolete* - Missing kernel start address
KU_ERROR_LAUNCHADDR	- *obsolete* - Missing kernel launch address

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.

Products are not designed, intended, or authorised 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 unauthorised 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 unauthorised use, even if such claim alleges that F&S Elektronik Systeme was negligent regarding the design or manufacture of said product.

Listings

Figures

Tables