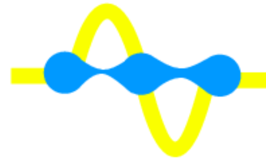


DSP-Weuffen

USB-UHPI-Programmer

DLL Programmers Manual

V1.01 – 2007-08-06

Authors: Alexander Stohr

Siegbert Baude

This page intentionally left blank

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

Content

1 Purpose of this documentation 6

2 DLL-Interface 7

2.1 Initialization and control routines 7

2.1.1 Debug outputs 7

2.1.2 Device opening and closing 8

2.2 Monolithic functions 8

2.2.1 COFF Conversion 9

2.2.2 RAM programming 10

2.2.3 FLASH programming 10

2.2.4 DSP hardware reset 11

2.2.5 Interrupt line state 11

2.3 UHPI functions 11

2.3.1 Control Word Access 11

2.3.2 Address latch access 12

2.3.3 Data transfers 13

2.4 Feature functions 14

2.4.1 Memory fill 14

2.4.2 Memory compare 14

3 Appendix 15

3.1 Abbreviations 15

3.2 Errata 15

Figures

Figure 1: Loading application data into RAM 10

Figure 2: Loading application data into FLASH-ROM 11

Figure 3: Relation between adressing formats 12

Tables

Table 1 Changes 4

Table 2 Used documents and references 5

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

Changes

Ver.	Date	Description	Executer
V0.01 Draft	2006-07-17	Start of Document	A. Stohr
V0.02 Draft	2006-07-17	Formatting	A. Stohr
V0.03 Draft	2006-08-11	Added description GetIntState, Fill32 and Compare32	A. Stohr
V0.04 Draft	2006-08-28	Split address functions into two flavors and figure.	A. Stohr
V1.00 Release	2006-10-26	Format changed, spell-checked	S. Baude
V1.01 Release	2007-08-06	Updated for COFF extended DLL and tools version 1.7	A. Stohr

Table 1 Changes

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

Used Documents and References

You will find the referenced documents in the “Documentation” folder on the CD.

Pos.	Description
1	TI sprs268d.pdf *TMS320C6727 [...] Floating-Point Digital Signal Processors* (Rev. D, February 2006)
2	TI spraa69c.pdf and sprc203.zip *Using the TMS320C672x Boot loader* (Rev. C, March 2006)
3	TI spru719.pdf *TMS320C672x DSP Universal Host Port Interface (UHPI)* (December 2005)
4	TI spra999a.pdf and spra999a.zip *Application Report: Creating a Second-Level Boot loader for FLASH Boot loading on TMS320C6000 Platform With Code Composer Studio* (Rev. A1, May 2006) Attention: This only covers C620x/670x and C621x/671x/64x devices, but not yet the C6727
5	TI spru186p.pdf *TMS320C6000 Assembly Language Tools v 6.0 Beta User’s Guide* (Rev. P, July 2005)
6	TI spru187n.pdf *TMS320C6000 Optimizing Compiler v 6.0 Beta* (Rev. N, July 2005)
7	http://focus.ti.com/docs/prod/folders/print/tms320c6727.html *TMS320C6727, Floating-Point Digital Signal Processor*
8	DSP-Weuffen EVM-board_Technical_Reference_TMS320C6727_V1_11.pdf *C6727 EVM, Technical Reference*

Table 2 Used documents and references

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

1 Purpose of this documentation

This document describes the DLL interface as exported by the USB-UHPI-Programmer device for the "C6727 EVM" board from the DSP-Weuffen GmbH. The software is intended to be used on Windows 2000 and Windows XP platforms in combination with standard development tools like Microsoft Visual C++ 6.0 or Microsoft Visual Studio Express and comparable products from third parties. Developers of alternate programming systems (e.g. Visual Basic or MatLAB) should be able to make use of this interface as well even if this document will not cover any specifics for their tool set.
This documentation is designed for the advanced programmer familiar with C/C++ programming including advanced technologies like the usage of third party libraries. Some basic knowledge in communications, DSP technology and flash memory programming might be helpful, but is not required. It is further highly recommended to get a general idea of the UHPI communication basics. See the related documentation for this.

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

2 DLL-Interface

The package consists of a DLL that should either be in the same path as the user application or in a central DLL repository of the Windows operating system (see http://msdn.microsoft.com). The project must specify another lib file for inclusion in the library path of the build system. Simply including the provided header files in the desired source files does the embedding within the C/C++ source code. For this purpose, the include paths should cover the directory where those header files are located or they have to be in the same directory as the sources. Multiple inclusions of header files are handled gently even if it's not good coding style at all.

The DLL can only drive a single interface per machine. If multiple interfaces are connected, only the very first will get opened and accessed. This prevents the need for the programmer to spend any time for a possibly complex device selection interface. The DLL should nicely work in any single and multi-threaded application whilst its entry points should not be called by more than one thread at the very same time. If called from multiple threads, the application programmer should take care that all accesses on this interface will get serialized.

2.1 Initialization and control routines

There are a few pretty basic functions that allow general info and control of the interface and the attached device.

2.1.1 Debug outputs

For troubleshooting purposes there is a small set of functions that allow control of verbosity.

```
void UHPI_SetDebugIfNull (void);
```

When called, any possible debug outputs of the DLL will get disabled. This is the default state of the DLL.

```
void UHPI_SetDebugIfCLI (void);
```

When called, the debug outputs of the DLL will get printed to the stdout channel, which most often means the commandline. If stdout channel is redirected, the prints will go this way. If there is no commandline then there will be no visible outputs.

```
#ifdef __ATLCTRLS_H__
void UHPI_SetDebugIfLB (CListBox *pListBox);
#endif
```


Copyright 2006 DSP-Weuffen GmbH, all rights reserved

If running as a MFC dialog based GUI application with ATL controls or something similar, it is possible to call this function with a pointer on the desired listbox object as parameter in order to direct the debugging outputs to this GUI element. If the given listbox has to be destroyed, the debugging should either be redirected to some other target or disabled prior to the given object being deleted and getting invalid.

2.1.2 Device opening and closing

For connecting to the device on demand and disconnecting again there is a set of matching functions.

```
bool UHPI_Open (void);
```

When this function is called the UHPI programmer interface is searched on the USB busses of the PC. If found, it will be openend so that it is usable for subsequent calls. In case of success “true” is returned. On any error the device is closed and “false” is returned.

```
bool UHPI_Close (void);
```

This call closes the currently open device. At present in any case “true” is returned. Future versions might be able to return “false”, if an error condition occured.

2.2 Monolithic functions

There are some rather big monolithic functional blocks that expose a programmable interface for the basic functionality of the programmer interface. The functions are comparable with what is provided as GUI and CLI applications in the programmer package.

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

2.2.1 COFF Conversion

```
typedef struct
{ unsigned char boot_loader_section[128];
  uint32_t boot_section_address;
  uint32_t boot_table_address;
  uint32_t entry_point;
} coff_info_t;
```

This structure defines a memory layout that holds several data elements for helping in the COFF to tektronix file format conversion process needed for allowing to upload a target binary that comes in the standard TI Code Composer output file format. Please note that the hex6x utility does provide an alternat way for generating your own tektronix formatted outputs. As this tool is a rather universal design it can offer you much more flexibility especially when it comes to segmented flash memory.

The structure element boot_loader_section contains a zero terminated single byte c string that defines the name of the section to use for the boot loader. With the boot_section_address you will define the start address where your flash memory has to offer the boot loader code. The boot_table_address defines where the start of the table for the sections to load will reside. With the entry_point address the self determined application load address gets reported.

```
bool UHPI_SetCoffInfoDefaults (coff_info_t *pcoff_info);
```

When this function is called the coff info structure that the parameter points to will get filled with the DLL built in defaults. In case of success “true” is returned. On error “false” is returned.

```
bool UHPI_ConvertCoff2Tektronix (const char *coffname, char *texname,
uint32_t texname_buflen, coff_info_t *pcoff_info);
```

This function is meant just as a helper for any sort of testing. It will read the coff file with the provided name, check it and convert it to a flash image. After that it will write a second file using the Tektronix file format. The output file name gets generated from the initial filename by replacing its extension and the name will get returned trough the provided buffer and its length value. The whole conversion will use the provided coff info structure for operation and will update the entry point member.

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

2.2.2 RAM programming

```
bool UHPI_ProgramTektronix (char *filename);
bool UHPI_ProgramCoff (const char *coffname, coff_info_t *pcoff_info);
```

When called the specified file is opened, loaded into PC memory and sent to the target RAM. In turn, first a hardware reset of the target gets performed. If the file is in Tektronix file format it will be used immedeately. For coff file format it will receive an instant conversion using the provided coff information. The file must contain an application with parallel flash boot header data. In case of success “true” is returned. On any error the device is closed and “false” is returned.

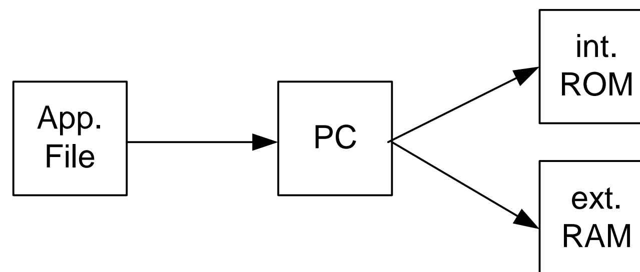


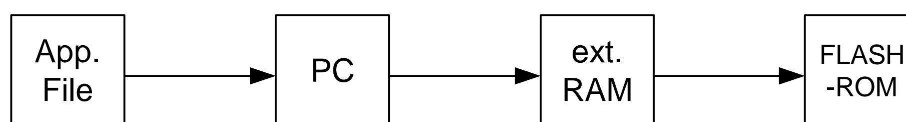
Figure 1: Loading application data into RAM

2.2.3 FLASH programming

```
bool UHPI_FlashTektronix (char *filename);
bool UHPI_FlashCoff (const char *coffname, coff_info_t *pcoff_info);
```

Attention: Before calling this routine a suitable flash programmer binary for the target must have been loaded into RAM and started. This is best done with the function `UHPI_ProgramTektronix`.

The given filename should identify a parallel flash image, which will get loaded into PC memory. After that it is sent to the external RAM of the target system. Having done this marks the start of the flash programming, which first erases the onboard flash memory and then flashes the still volatile RAM based data into the permanent flash ROM of the target. The provided image has to be encoded in Tektronix file format containing a parallel flash boot header aside to your application program. In case of success “true” is returned. On any error the device is closed and “false” is returned.



Copyright 2006 DSP-Weuffen GmbH, all rights reserved

Figure 2: Loading application data into FLASH-ROM

2.2.4 DSP hardware reset

```
bool UHPI_DspHwReset (void);
```

When called a hardware reset of the target gets performed and the target access through UHPI gets re-initialized. In case of success "true" is returned. On any error the device is closed and "false" is returned.

2.2.5 Interrupt line state

```
bool UHPI_GetIntState (bool *pState);
```

Querys the state of the interrupt line as received by the programming interface. A return value of "false" for *pState means nRDY is in low state thus an interrupt is pending. A return value of "true" for *pState means nRDY is in high state thus no interrupt is pending. In case of success "true" is returned. On any error the device is closed and "false" is returned. For interrupt control see the TI documentation.

2.3 UHPI functions

The UHPI provides a basic set of control and memory/data access functions. These can be sub-divided into two directions of access (one is read, the other is write) and into four categories of target object operation (control register, address register, memory/data access and memory/data access with auto-increment). Details on these registers are subject of the TI documentation for the target.

2.3.1 Control Word Access

```
bool UHPI_SetControlWord (uint32_t control);
```

Sets the given UHPI control word for the target. In case of success "true" is returned. On any error the device is closed and "false" is returned.

```
bool UHPI_GetControlWord (uint32_t *pcontrol);
```

Reads the UHPI control word from the target into the provided buffer. In case of success "true" is returned. On any error the device is closed and "false" is returned.

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

2.3.2 Address latch access

Please note the significant difference in between the adress word value encoding in the target address latch and a linear address encoding as it is common to the DSP programmer.

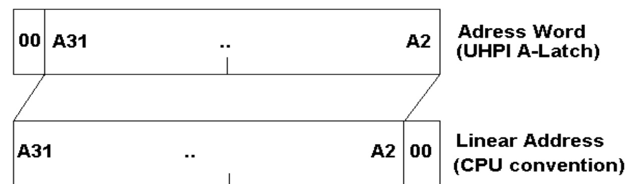


Figure 3: Relation between adressing formats

```
bool UHPI_SetAddressWord (uint32_t address);
```

Sets the given UHPI adress value for the targets adress latch. The encoding is done as in the target address latch. In case of success “true” is returned. On any error the device is closed and “false” is returned.

```
bool UHPI_SetLinearAddress (uint32_t address);
```

Sets the given UHPI adress value for the targets adress latch. The encoding is done as in the CPU context of the DSP. In case of success “true” is returned. On any error the device is closed and “false” is returned.

```
bool UHPI_GetAddressWord (uint32_t *paddress);
```

Reads the UHPI address value from the target latch into the provided buffer. The encoding resembles the address latch. In case of success “true” is returned. On any error the device is closed and “false” is returned.

```
bool UHPI_GetLinearAddress (uint32_t *paddress);
```

Reads the UHPI address value from the target latch into the provided buffer. The encoding resembles the address encoding as it is used by the CPU core of the DSP. In case of success “true” is returned. On any error the device is closed and “false” is returned.

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

2.3.3 Data transfers

The UHPI provides a set of read and write functions that allow accessing of most of the logical units in the memspace of the target. In fact all RAM areas are accessible by that set of functions.

```
bool UHPI_SetData (uint32_t count, const uint32_t *pBuf);
```

Sets the given UHPI data for the target at the latched adress value. In case of success “true” is returned. On any error the device is closed and “false” is returned.

```
bool UHPI_GetData (uint32_t count, uint32_t *pBuf);
```

Reads the UHPI data value at the latched adress value from the the target into the provided buffer. In case of success “true” is returned. On any error the device is closed and “false” is returned.

```
bool UHPI_SetDataIncr (uint32_t count, const uint32_t *pBuf);
```

Sets the given UHPI data for the target at the latched adress value. The adress latch is incremented after the access. In case of success “true” is returned. On any error the device is closed and “false” is returned.

```
bool UHPI_GetDataIncr (uint32_t count, uint32_t *pBuf);
```

Reads the UHPI data value at the latched adress value from the the target into the provided buffer. The adress latch is incremented after the access. In case of success “true” is returned. On any error the device is closed and “false” is returned.

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

2.4 Feature functions

There are functions that allow actions to be carried out for the programmer and the target. Currently the only feature function available is letting the UHPI interface semi-autonomously fill up a certain amount of memory on the target with a specified value.

This category might see some extensions in future depending on special customer needs and vendor decisions. On customer demand DSP-Weuffen GmbH is able to offer additional services that fulfill a much wider range of features with the provided interface hardware.

2.4.1 Memory fill

```
bool UHPI_FeatureMemoryFill32 (uint32_t value, uint32_t bytes);
```

Repeatedly writes the given value to the current memory location of the target using the adress auto increment mode. The number of bytes must be a multiple of 4, which is the word size. In case of success “true” is returned. On any error the device is closed and “false” is returned.

2.4.2 Memory compare

```
bool UHPI_FeatureMemoryCompare32 (uint32_t value, uint32_t bytes,
bool *pequals);
```

Repeatedly reads the given amount of bytes from the target system using the current address value and address auto increment mode. Any read word is compared with the provided compare value. The number of bytes must be a multiple of 4, which is the word value. The comparison result is reported through the return parameter *pequals and is “true” if all values are equal or “false” if a value was found that does differ. The checking will stop on the first detected difference. In case of success “true” is returned. On any error the device is closed and “false” is returned.

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

3 Appendix

3.1 Abbreviations

DSP	Digital Signal Processor
TI	Texas Instruments
ROM	Read Only Memory, non-volatile
FLASH	A ROM, but programmable only in larger blocks after an explicit erase
RAM	Random Access Memory
SDRAM	Synchronous Dynamic RAM
UHPI	Universal Host Port Interface
EMIF	Extended Memory Interface
PLL	Phase Locked Loop
IRQ	Interrupt Request
USB	Universal Serial Bus
MCP	Microprocessor
EVM	Evaluation Module
IDE	Integrated Development Environment
CCS	Code Composer Studio (TI IDE)
GUI	Graphical User Interface
CLI	Command Line Interface
IDE	Integrated Drive Electronics
PCB	Printed Circuit Board
DLL	Dynamically Linkable Library

3.2 Errata

No known errors yet.

Copyright 2006 DSP-Weuffen GmbH, all rights reserved

This page intentionally left blank

Copyright 2006 DSP-Weuffen GmbH, all rights reserved