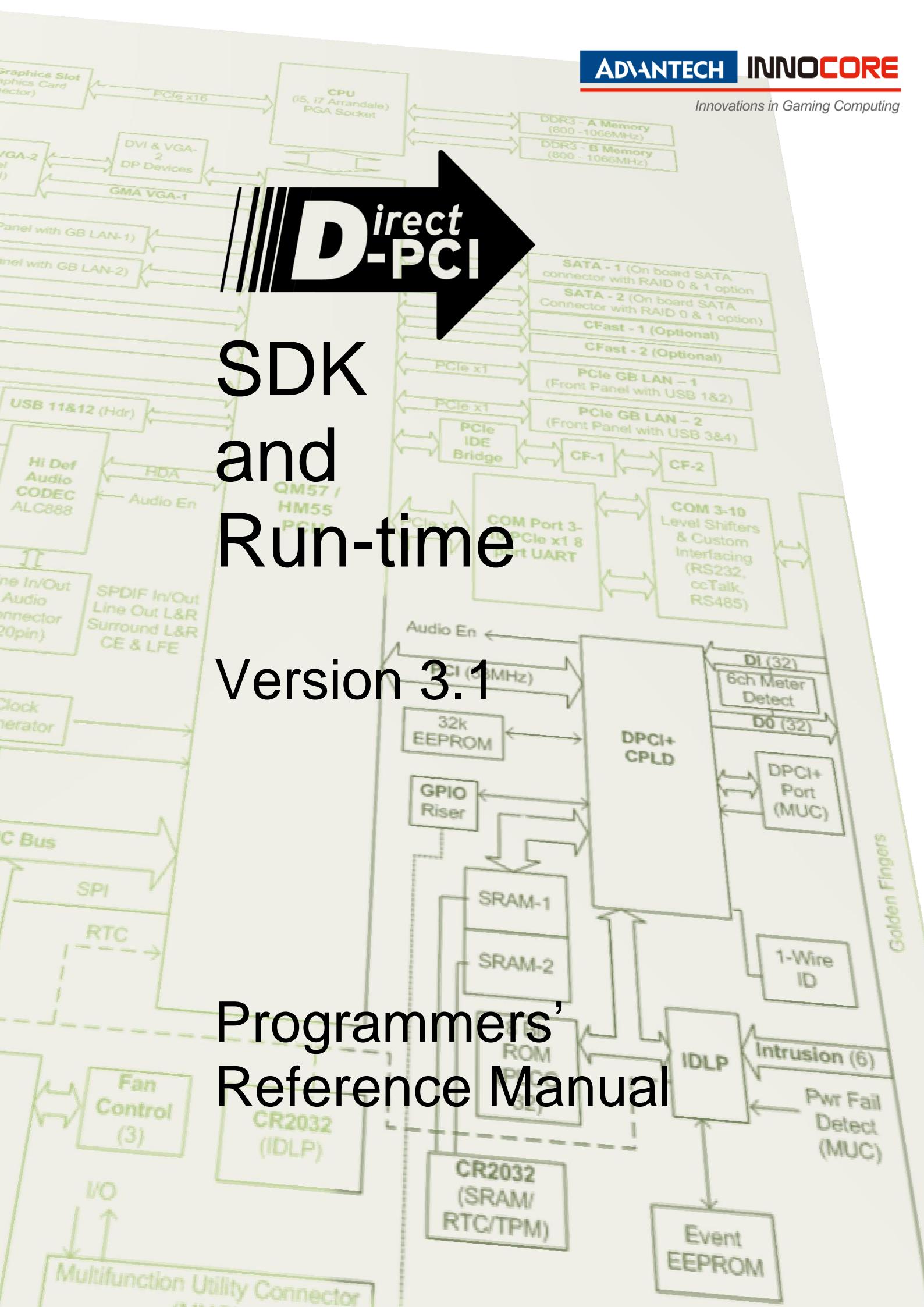




# SDK and Run-time

Version 3.1

## Programmers' Reference Manual



Page Intentionally Blank

## Revision History

<u>Version</u>	<u>Author</u>	<u>Date</u>	<u>Notes</u>
1.0	AD	25/8/05	Initial release imported from 50-1501
1.1	AD	26/9/05	Updates for new API functions in 1.2.0 release
1.2	AD	21/2/06	Updates for 1.2.1 release
1.3	AD/SC	22/2/06	Release for issue
1.4	AD	14/12/06	Updates for 1.3.0 release
1.5	AD		Updates for 1.4.0 release (changed from 206-087)
2.0	AD/MC	01/05/08	Updates for v2.0.0 release
2.01	AD/MC	10/06/08	More updates for v2.0.0
2.02	AD/MC	23/10/08	Few more updates for v2.0.0: changed text on 32-bit PCI accesses to SRAM.
2.03	MC	6/11/08	Updated return values for dpci_bat_get_status(), included dpci_idprom_readdir().
2.04	AD	19/12/08	Fix dpci_bat_read() description and place in manual
2.05	AD	8/4/09	Fix table of i2c support; layout/format updated to latest design.
2.06	MC	07/05/09	Updates for DPCI v2.2.0
2.07	AD	07/04/10	Added comments on thread-safety in 1.3.1 and (bug 213) and SRAM API (2.1) and SRAM map/unmap functions Updated availability notes on sram/rom map/unmap functions. Fix name of dpci_id_setsleeptime and description/notes in 2.5.6. Document dpci_id_select_event() and –L option to idutil (bug 227)
2.08	MC	02/07/10	Add new API dpci_bios_dump()
3.0	AD/MC	02/07/10	Updates for 3.0.0 release: new event-streaming API added.
3.01	AD/MC	25/08/10	Rework of manual introduction section. Dpci_io_{read,write}{8,16,32}() functions now marked deprecated.
3.02	AD/MC/EM	27/08/10	Note about removed platforms: DPX-112, DPX-116(u), DPX- 117, 80-1003; example updated for newer platforms in respect of older removed platforms. Fix bug 275: description of timeout_ms parameter to dpci_id_wait_event() is wrong. Fix bug 198: poor instructions on how to build demo programs. Fix bug 231: no instructions how to use SRAM as disk drive Fix bug 291: references to dpci_demo.c which is renamed io_demo.c Fix bug 290: documentation retained for removed function dpci_io_config_input0(). Fix bug 289: offset parameter type for SRAM/ROM API calls was not listed correctly Further reworks of the manual's introduction.

			Fixed many cross references Added table of figures Better formatting of information relating header files, driver files etc. for each API. Quiet-mode and I/O board APIs get their own sections. Add descriptions for missing functions: dpci_idprom_read(), dpci_idprom_read8(), dpci_iowd_pat(), dpci_iowd_enable() and dpci_iowd_disable().
3.03	AD/EM	05/02/11	Corrections for spelling and grammar
3.04	AD/EM	15/02/11	More and better cross referencing Further corrections for spelling and grammar and typological inconsistencies; Host watch-dog functions given own section of manual Updated illustrations of importing DirectPCI embedded components Added availability sections for host watchdog functions. Added description of dpci_id_fwversion_full(). Figure references given to tables and structure member information.
3.05	AD/EM	18/02/11	Add sections for missing gpio functions Fix spurious blank pages Correct description of parameters to register and unregister callback functions Correct description of how event filtering occurs w.r.t. IDLP events when registering a callback. Fix page breaks Fix document map Enable PDF bookmarks Fix bug 328: unclear what port numbers to use for dpci_io_{read,write}_port() functions.
3.06	AD/MC	08/08/11	Section 2.15.1 – update quiet mode support on DPX-S, C and E-series motherboards Section 2.17 - Fix some typing errors in GPIO API reference Section 2.6 and 2.16 - Fix bug 360: Document the operation of I/O WDT on standalone boards. Section 2.18.3 - Fix bug 384: No documentation for dpci_bios_dump(). Section 2.5.1 Fix bug 259: Timestamp returned in dpci_event by event_wait APIs is not consistent across OSs. Section 2.5 - Add code examples for dpci_ev_wait() and dpci_ev_wait_all(). Sections 2.9.3 and 2.9.4: fix bug 273: not clear what mapping of ports to DI/DO lines to golden finger pins is.
3.07	AD	16/03/12	Fix page footer to show correct version of document. Fix company addresses. Update prerequisites (scn 1.5.3). Removed section 1.6 (General Notes) and moved non-duplicated information to other relevant sections for Windows and Linux

3.08	AD	16/06/12	<p>Corrected number of parameters to printf in example for dpci_drv_version().</p> <p>Correct description of dpci_bat_set_check_period so as to include better description of the code parameter.</p>
3.09	KY	30/10/12	UK Postcode Errata
3.0_10 (3.10)	RS	03/01/13	<p>Version number updated for readability and section 1.10 explaining the nature of the change.</p> <p>Improved documentation of dpci_bat_set_check_period() and dpci_bat_get_level(). Improved clarity of the documentation for the batt command (section 4.1).</p> <p>Added documentation for dpci_core_get_serial()</p> <p>Stylistic updates for consistency across the product range.</p>
3.1_1	AD	9/10/13	Manual for v3.1.x.
3.1_2	AD/JC	30/4/14	<p>Fix bug 718: "Explain further the implications of access alignment limitatons." Added section documenting which boards are affected. Corrected typo on title page.</p> <p>Fix bug 622 "Documentation should indicate that sram/rom demo need Administrator privilege". Updated scn.2.1 to explain required changes to VisualStudio projects.</p> <p>Reflect changes in IDLP f/w v60 in section 2.7 (IDLP API). Update the list of events and descriptions, explain more about common errors from IDLP commands.</p>
3.1_3	AD	05/03/2015	<p>Updated section 1.3 to reflect recent changes to list of deprecated or removed functions.</p> <p>Added new section (1.9) documenting the facility to chang or /remove SRAM/ROM device drive letters.</p>
3.1_4	AD	08/07/2015	<p>Add function dpci_bios_size().</p> <p>Added many corrections to cross references.</p> <p>New explanations regarding serial drivers installation.</p>
3.1_5	AD/AM	07/08/2015	<p>Fix description of dpci_bios_size().</p> <p>Remove text about restricted access to first 256 bytes of motherboard EEPROM (API section 2.11 ).</p> <p>Update section 2.1 regarding gaming functions and how the SDK supports them.</p> <p>Add full documentation for IDPROM API (section 2.9 ).</p> <p>Add new section 2.21 documenting the debug API.</p> <p>Add new section 2.22 documenting the boards API.</p> <p>Augmented section 1.7 regarding debugging information.</p> <p>Fix bug 893: "dpci_bat_get_status returns status mask instead of 1 when battery is OK" in section 2.14.7.</p> <p>Updated text example output from batt command in section 4.1 .</p> <p>Added new section 2.21 to describe debug API added in v3.1.1.</p> <p>Inserted new section 3.1 regarding support for Microsoft Windows Embedded 7/8.</p>

Page Intentionally Blank

© 2016 ADVANTECH Co LTD

This document may not be copied, reproduced, translated, transmitted or reduced to any printed or electronic medium or to any machine-readable form, or stored in a retrieval system, either in whole or in part, without the prior written consent of **ADVANTECH CORP.**

Whilst every effort has been made to ensure the accuracy of this document, **ADVANTECH CORP** accepts no responsibility for any errors or omissions contained herein. However, if you do discover any inaccuracy in this document, we would be grateful if you would forward the details to [support@advantech-innocore.com](mailto:support@advantech-innocore.com).

**ADVANTECH INNOCORE** is a trade name of **ADVANTECH Co. LTD.**

Contact us at :

Advantech Innocore  
Innocore House  
Kingfisher Way  
Silverlink Business Park,  
Newcastle upon Tyne NE28 9NX  
Tel: +44 (0)191 2624844  
Fax: +44 (0)191 2639287  
[www.advantech-Innocore.com](http://www.advantech-Innocore.com)  
Email [sales@advantech-Innocore.com](mailto:sales@advantech-Innocore.com)

Page Intentionally Blank

# Contents

<b>REVISION HISTORY .....</b>	<b>3</b>
<b>1 INTRODUCTION .....</b>	<b>19</b>
<b>1.1 OVERVIEW.....</b>	<b>19</b>
<b>1.2 SUPPORT .....</b>	<b>21</b>
1.2.1 HARDWARE PLATFORMS.....	21
1.2.2 HARDWARE PLATOFMRS NO LONGER SUPPORTED.....	21
1.2.3 OPERATING SYSTEMS: MICROSOFT WINDOWS.....	21
1.2.4 OPERATING SYSTEMS: LINUX .....	22
1.2.5 CUSTOMER SUPPORT .....	22
1.2.6 THREAD-SAFETY .....	22
1.2.7 .NET SUPPORT .....	22
1.2.8 .NET DEMO PROGRAMS.....	23
<b>1.3 FEEDBACK .....</b>	<b>24</b>
<b>1.4 CHANGES IN THE DIRECTPCI SDK &amp; RUN-TIME V3.1 .....</b>	<b>24</b>
<b>1.4.1 DEPRECATED FEATURES .....</b>	<b>25</b>
<b>1.5 WINDOWS INSTALLATION .....</b>	<b>27</b>
<b>1.5.1 OBTAINING THE CORRECT INSTALLER FILES .....</b>	<b>27</b>
<b>1.5.2 RELEASE NOTES AND CHANGE-LOG .....</b>	<b>27</b>
<b>1.5.3 PREREQUISITES .....</b>	<b>27</b>
<b>1.5.4 RUNNING THE INSTALLER.....</b>	<b>28</b>
<b>1.5.5 CHOOSING WHAT TO INSTALL .....</b>	<b>28</b>
<b>1.5.6 INSTALLING THE RUN-TIME DEVICE-DRIVERS AND LIBRARIES.....</b>	<b>31</b>
<b>1.5.7 SHARED AND STATIC LIBRARIES .....</b>	<b>32</b>
<b>1.5.8 CHANGING FROM 80-1003 TO 80-0062 I/O BOARDS ON MICROSOFT WINDOWS SYSTEMS.....</b>	<b>32</b>
<b>1.5.9 SRAM/ROM DRIVER INSTALLATION NOTE.....</b>	<b>32</b>
<b>1.5.10 SRAM &amp; ROM DEVICES IN DISK-DRIVE (BLOCK) MODE.....</b>	<b>33</b>
<b>1.5.11 REMOVING THE PRODUCT .....</b>	<b>33</b>
<b>1.6 LINUX INSTALLATION .....</b>	<b>35</b>
<b>1.6.1 OBTAINING THE CORRECT INSTALLER FILES .....</b>	<b>35</b>
<b>1.6.2 RELEASE NOTES AND CHANGE-LOG .....</b>	<b>35</b>
<b>1.6.3 PREREQUISITES .....</b>	<b>35</b>
<b>1.6.4 INSTALLATION.....</b>	<b>36</b>
<b>1.6.5 UNATTENDED INSTALLATION .....</b>	<b>36</b>
<b>1.6.6 LINUX KERNEL UPDATES .....</b>	<b>36</b>
<b>1.6.7 LINUX BOOTING.....</b>	<b>37</b>
<b>1.6.8 UDEV SUPPORT .....</b>	<b>37</b>
<b>1.6.9 LINUX SERIAL PORTS .....</b>	<b>37</b>
<b>1.6.10 SHARED AND STATIC LIBRARIES .....</b>	<b>38</b>
<b>1.7 DEBUGGING .....</b>	<b>39</b>
<b>1.7.1 DEBUG AND RELEASE BINARIES.....</b>	<b>39</b>
<b>1.7.2 WINDOWS .....</b>	<b>39</b>
<b>1.7.3 LINUX .....</b>	<b>39</b>
<b>1.7.4 CORE I/O DRIVER DEBUG LEVELS .....</b>	<b>40</b>
<b>1.7.5 DIRECTPCI API LIBRARY DEBUG LEVELS .....</b>	<b>41</b>
<b>1.8 UTILITIES.....</b>	<b>44</b>
<b>1.9 DEMO PROGRAMS AND CODE SAMPLES .....</b>	<b>45</b>

1.9.1 DEMONSTRATION CODE UNDER LINUX .....	45
1.9.2 DEMONSTRATION CODE UNDER MICROSOFT® WINDOWS® XP.....	47
<b>1.10 CHANGING THE DRIVE LETTERS FOR SRAM AND ROM DRIVES .....</b>	<b>48</b>
1.10.1 PROCEDURE FOR CHANGING THE DRIVE LETTER .....	48
1.10.2 NOTE ABOUT DELAYS WHEN RESUMING/WAKING A SYSTEM WITH AN UNFORMATTED SRAM OR ROM DRIVE .....	49
<b>1.11 NOTE ON SOFTWARE VERSION AND RELEASES .....</b>	<b>50</b>
<b>1.12 NOTE ON DOCUMENTATION VERSIONS .....</b>	<b>51</b>
<b>2 THE DIRECTPCI APIs.....</b>	<b>53</b>
<b>2.1 WHICH API SUPPORTS WHICH GAMING FUNCTION?.....</b>	<b>53</b>
2.1.1 APIs GROUPED BY GAMING FUNCTION .....	53
2.1.2 STORAGE MEDIA ACCESSIBLE USING DIRECTPCI APIs .....	54
2.1.3 HANDLING DIGITAL I/O AND DOOR SWITCHES .....	56
2.1.4 COMPLETE LIST OF ALL DIRECTPCI APIs .....	57
<b>2.2 SRAM API.....</b>	<b>61</b>
2.2.1 FILES FOR THE SRAM API .....	61
2.2.2 SRAM DISK DRIVE.....	64
2.2.3 GET SIZE OF SRAM .....	66
2.2.4 READ BYTE FROM SRAM.....	67
2.2.5 READ WORD FROM SRAM.....	68
2.2.6 READ DWORD FROM SRAM .....	69
2.2.7 READ BLOCK FROM SRAM .....	70
2.2.8 WRITE BYTE TO SRAM.....	72
2.2.9 WRITE WORD TO SRAM .....	73
2.2.10 WRITE DOUBLE WORD TO SRAM .....	74
2.2.11 WRITE BLOCK TO SRAM.....	75
2.2.12 MEMORY MAP SRAM.....	76
2.2.13 UNMAP SRAM .....	78
2.2.14 MEMORY MAPPED SRAM (LINUX ONLY).....	79
<b>2.3 ROM API.....</b>	<b>81</b>
2.3.1 FILES FOR THE ROM API .....	81
2.3.2 GET SIZE OF ROM .....	83
2.3.3 READ BYTE FROM ROM .....	85
2.3.4 READ WORD FROM ROM .....	86
2.3.5 READ DWORD FROM ROM .....	87
2.3.6 READ BLOCK FROM ROM .....	88
2.3.7 MEMORY MAP ROM.....	90
2.3.8 UNMAP ROM .....	92
2.3.9 MEMORY MAPPED ROM (LINUX ONLY) .....	93
<b>2.4 DIRECTPCI CORE I/O DRIVER .....</b>	<b>95</b>
2.4.1 FILES FOR THE CORE API.....	95
<b>2.5 SOFTWARE VERSION MANAGEMENT API .....</b>	<b>97</b>
2.5.1 GET API VERSION CODE .....	97
2.5.2 GET API VERSION STRING .....	99
2.5.3 GET DRIVER VERSION CODE .....	100
2.5.4 GET DRIVER VERSION STRING.....	102
2.5.5 GET HARDWARE VERSION/REVISION .....	103
2.5.6 GET DRIVER HARDWARE PROFILE .....	104
<b>2.6 EVENT STREAMING API .....</b>	<b>105</b>
2.6.1 DPCI_EVENT_T .....	106

2.6.2	WAIT FOR EVENTS OF ALL TYPES .....	108
2.6.3	WAIT FOR SPECIFIC EVENT TYPES .....	110
2.6.4	REGISTER A CALLBACK .....	113
2.6.5	UNREGISTER A CALLBACK .....	115
2.6.6	SET INPUT DE-BOUNCE PARAMETERS .....	116
<b>2.7</b>	<b>HOST WATCHDOG API.....</b>	<b>118</b>
2.7.1	ENABLE IDLP HOST WATCHDOG .....	118
2.7.2	DISABLE IDLP HOST WATCHDOG .....	120
2.7.3	DPCI WATCHDOG FUNCTIONS: WATCHDOG RESET .....	121
<b>2.8</b>	<b>INTRUSION DETECTION AND LOGGING PROCESSOR API .....</b>	<b>123</b>
2.8.1	ERROR CODES COMMON TO ALL IDLP API ENTRIES .....	123
2.8.2	GET IDLP FIRMWARE VERSION .....	124
2.8.3	GET FULL IDLP FIRMWARE VERSION .....	125
2.8.4	GET IDLP INTRUSION LINE STATUS .....	127
2.8.5	SET IDLP SLEEP PERIOD .....	129
2.8.6	GET NUMBER OF UNREAD IDLP EVENTS .....	131
2.8.7	READ AN IDLP EVENT .....	132
2.8.8	READ AN OLD IDLP EVENT.....	137
2.8.9	WAIT FOR AN IDLP EVENT.....	139
2.8.10	GET NAME FOR IDLP EVENT NUMBER .....	142
2.8.11	SET IDLP DATE AND TIME .....	145
2.8.12	GET IDLP DATE AND TIME.....	147
2.8.13	FIND THE LAST TIME AN EVENT OCCURRED .....	148
2.8.14	OBTAIN THE IDLP FIRMWARE CHECKSUM .....	150
<b>2.9</b>	<b>IDPROM API.....</b>	<b>151</b>
2.9.1	GET IDPROM ID .....	151
2.9.2	GET SIZE OF IDPROM .....	152
2.9.3	READ BYTE FROM IDPROM .....	153
2.9.4	WRITE BYTE TO IDPROM .....	154
2.9.5	READ BLOCK FROM IDPROM .....	156
2.9.6	WRITE BLOCK TO IDPROM .....	158
<b>2.10</b>	<b>DIGITAL I/O API .....</b>	<b>160</b>
2.10.1	GET NUMBER OF DIGITAL INPUT PORTS .....	160
2.10.2	GET NUMBER OF DIGITAL OUTPUT PORTS .....	162
2.10.3	READ DIGITAL INPUT PORT .....	163
2.10.4	WRITE DIGITAL OUTPUT PORT .....	165
2.10.5	CHANGE DIGITAL OUTPUT PORT .....	167
2.10.6	READ DIGITAL OUTPUT PORT .....	169
2.10.7	WAIT FOR AN EDGE-TRIGGERED INTERRUPT ON A DIGITAL INPUT PORT .....	170
2.10.8	WAIT FOR AN EDGE-TRIGGERED INTERRUPT ON DIGITAL INPUT PORT #0 (DIN0-7) .....	173
2.10.9	WAIT FOR AN INTERRUPT ON DIGITAL INPUT PORT .....	174
2.10.10	WAIT FOR AN INTERRUPT ON ALL DIGITAL INPUT PORTS .....	176
2.10.11	EDGE-TRIGGERED DIGITAL INPUT EVENTS (LINUX ONLY, DEPRECATED) .....	178
2.10.12	READ BYTE FROM DIRECTPCI REGISTER (DEPRECATED) .....	181
2.10.13	READ WORD FROM DIRECTPCI REGISTER (DEPRECATED) .....	182
2.10.14	READ DOUBLE WORD FROM DIRECTPCI REGISTER (DEPRECATED) .....	183
2.10.15	WRITE BYTE TO DIRECTPCI REGISTER (DEPRECATED) .....	184
2.10.16	WRITE WORD TO DIRECTPCI REGISTER (DEPRECATED) .....	185
2.10.17	WRITE DOUBLE WORD TO DIRECTPCI REGISTER (DEPRECATED) .....	186
<b>2.11</b>	<b>MOTHERBOARD EEPROM API .....</b>	<b>187</b>
2.11.1	GET SIZE OF EEPROM.....	187
2.11.2	READ BYTE FROM EEPROM.....	188
2.11.3	WRITE BYTE TO EEPROM .....	190

2.11.4 READ BLOCK FROM EEPROM.....	192
2.11.5 WRITE BLOCK TO EEPROM.....	194
<b>2.12 I<sup>2</sup>C API .....</b>	<b>196</b>
2.12.1 GET NUMBER OF I <sup>2</sup> C BUSES .....	196
2.12.2 GET BUS NUMBER FOR A NAMED I <sup>2</sup> C BUS.....	197
2.12.3 GET I <sup>2</sup> C BUS NAME .....	198
2.12.4 I <sup>2</sup> C BUS ACCESS.....	200
2.12.5 TRANSLATE I <sup>2</sup> C ERROR CODE .....	205
2.12.6 RETURN LAST I <sup>2</sup> C ERROR CODE.....	206
2.12.7 RETURN LAST I <sup>2</sup> C ERROR STRING .....	207
2.12.8 GENERIC I <sup>2</sup> C READ ACCESS .....	208
2.12.9 GENERIC I <sup>2</sup> C WRITE ACCESS .....	212
2.12.10 8-BIT I <sup>2</sup> C READ ACCESS .....	214
2.12.11 8-BIT I <sup>2</sup> C WRITE ACCESS.....	216
<b>2.13 TEMPERATURE SENSOR API (I/O BOARD II).....</b>	<b>218</b>
2.13.1 TEMPERATURE SENSOR INITIALISATION .....	218
2.13.2 GET NUMBER OF TEMPERATURE SENSORS .....	219
2.13.3 GET BUS NUMBER FOR A NAMED TEMPERATURE SENSOR .....	220
2.13.4 GET TEMPERATURE SENSOR NAME .....	221
2.13.5 READ TEMPERATURE SENSOR.....	223
2.13.6 CONFIGURE TEMPERATURE SENSOR .....	225
2.13.7 READ TEMPERATURE SENSOR CONFIGURATION.....	227
2.13.8 CHECK TEMPERATURE SENSORS' STATES .....	229
2.13.9 WAIT FOR A TEMPERATURE SENSOR INTERRUPT.....	231
2.13.10 GET VALUE FOR A TEMPERATURE SENSOR STATE-NAME .....	233
2.13.11 GET TEMPERATURE SENSOR STATE-NAME .....	234
<b>2.14 BATTERY API.....</b>	<b>235</b>
2.14.1 GET NUMBER OF BATTERIES .....	235
2.14.2 CHECK BATTERY STATUS .....	236
2.14.3 GET BATTERY NAME .....	238
2.14.4 GET BATTERY INDEX NUMBER .....	239
2.14.5 GET BATTERY STATUS MASK .....	240
2.14.6 GET BATTERY VOLTAGE LEVEL.....	241
2.14.7 GET STATUS OF A BATTERY.....	242
2.14.8 SET BATTERY CHECK PERIOD .....	243
2.14.9 SET BATTERY ERROR LEVEL .....	245
2.14.10 GET BATTERY ERROR LEVEL.....	246
<b>2.15 ERROR HANDLING API.....</b>	<b>247</b>
2.15.1 GET LAST ERROR CODE.....	247
2.15.2 GET LAST ERROR STRING .....	248
2.15.3 GET ERROR STRING.....	249
<b>2.16 QUIET-MODE API.....</b>	<b>250</b>
2.16.1 GET QUIET MODE STATUS .....	250
2.16.2 SET QUIET MODE STATUS.....	252
<b>2.17 I/O BOARD API .....</b>	<b>253</b>
2.17.1 CHECK I/O BOARD SUPPORT .....	253
2.17.2 GET I/O BOARD IDENTIFIER .....	254
2.17.3 GET I/O BOARD REVISION .....	255
2.17.4 ENABLE I/O BOARD WATCHDOG .....	256
2.17.5 DISABLE I/O BOARD WATCHDOG .....	257
2.17.6 PAT I/O BOARD WATCHDOG .....	258
<b>2.18 GPIO API .....</b>	<b>259</b>

2.18.1 READ GPIO INPUT LINE .....	259
2.18.2 READ GPIO OUTPUT LINE.....	261
2.18.3 WRITE GPIO OUTPUT LINE .....	262
<b>2.19 DPX-E130 BACKUP POWER SUPPLY API.....</b>	<b>263</b>
2.19.1 ENABLE THE BPS FACILITY .....	263
2.19.2 CHECK BPS ENABLED STATUS.....	264
2.19.3 CHECK BPS IS CHARGED.....	265
<b>2.20 MISCELLANEOUS API .....</b>	<b>266</b>
2.20.1 READ DIP SWITCH STATUS .....	266
2.20.2 WAIT FOR PFD INTERRUPT .....	267
2.20.3 READ SYSTEM BIOS SIZE .....	268
2.20.4 READ SYSTEM BIOS MEMORY .....	269
2.20.5 GET SERIAL IDENTIFIER (DPX-C710 ONLY).....	270
<b>2.21 DEBUG API.....</b>	<b>271</b>
2.21.1 SET DPCI_CORE DRIVER DEBUG LEVEL .....	271
2.21.2 GET DPCI_CORE DRIVER DEBUG LEVEL .....	272
2.21.3 SET API LIBRARY DEBUG LEVEL .....	273
2.21.4 MODIFY API LIBRARY DEBUG LEVEL .....	274
2.21.5 GET API LIBRARY DEBUG LEVEL.....	275
2.21.6 SET API LIBRARY DEBUG MESSAGE CALLBACK .....	276
2.21.7 SET API LIBRARY DEBUG OUTPUT FILE .....	278
2.21.8 SET API LIBRARY DEBUG OUTPUT HANDLE .....	279
<b>2.22 BOARD IDENTIFICATION API.....</b>	<b>280</b>
2.22.1 GET THE BOARD HOST CODE .....	280
2.22.2 GET THE BOARD HOST NAME .....	282
2.22.3 GET THE BOARD HOST TAG.....	283
2.22.4 TRANSLATE A BOARD CODE NUMBER TO BOARD NAME .....	284
2.22.5 TRANSLATE A BOARD CODE NUMBER TO BOARD TAG .....	285
<b>3 WINDOWS EMBEDDED SUPPORT.....</b>	<b>287</b>
<b>3.1 WINDOWS EMBEDDED STANDARD 7, WINDOWS 8 EMBEDDED .....</b>	<b>287</b>
<b>3.2 WINDOWS XP EMBEDDED, WINDOWS EMBEDDED STANDARD 2009 .....</b>	<b>293</b>
3.2.1 COMPONENT INSTALLATION .....	293
3.2.2 COMPONENTS .....	295
3.2.3 ADVANTECH INNOCORE DIRECTPCI APPLICATION RUN-TIME .....	297
3.2.4 ADVANTECH INNOCORE DIRECTPCI DEMO PROGRAMS .....	298
3.2.5 ADVANTECH INNOCORE DIRECTPCI MULTI-FUNCTION ADAPTOR FOR 80-0062/A I/O BOARD ..	300
3.2.6 ADVANTECH INNOCORE DIRECTPCI MULTI-FUNCTION ADAPTOR FOR DPX-116U & DPX112..	301
3.2.7 ADVANTECH INNOCORE DIRECTPCI RUN-TIME AND DRIVERS .....	302
3.2.8 ADVANTECH INNOCORE DIRECTPCI SRAM AND ROM DRIVERS.....	303
3.2.9 ADVANTECH INNOCORE DIRECTPCI UTILITY PROGRAMS .....	304
<b>4 COMMAND LINE UTILITIES .....</b>	<b>305</b>
<b>4.1 BATT – SYSTEM BATTERY STATUS .....</b>	<b>305</b>
<b>4.2 DIPSW – DIP SWITCH STATUS .....</b>	<b>307</b>
<b>4.3 DPCI – GENERAL DIRECTPCI UTILITY .....</b>	<b>308</b>
<b>4.4 DPXNAME – SHOW THE NAME OF THE TARGET HARDWARE.....</b>	<b>311</b>
<b>4.5 IDUTIL – IDLP OPERATIONS .....</b>	<b>312</b>
<b>4.6 IOBOARD - I/O BOARD OPERATIONS .....</b>	<b>315</b>

<b>4.7 MEMUTIL – EEPROM, SRAM, ROM MEMORY ACCESS .....</b>	<b>317</b>
<b>APPENDIX 1 CHANGES IN PREVIOUS VERSIONS .....</b>	<b>321</b>
<b>A1.1 CHANGES IN V3.0.0 .....</b>	<b>321</b>
<b>A1.2 CHANGES IN V2.3.0 .....</b>	<b>322</b>
<b>A1.3 CHANGES IN V2.2.0 .....</b>	<b>322</b>
<b>APPENDIX 2 ADVANTECH WORLDWIDE .....</b>	<b>325</b>

Page Intentionally Blank

# LIST OF FIGURES

Figure 1.	Example API entry.....	20
Figure 2.	API Entries deprecated in v3.1.0 .....	25
Figure 3.	Windows Installation Types .....	27
Figure 4.	Splash-screen on running Setup.exe.....	28
Figure 5.	Types of Installation.....	28
Figure 6.	Window for Choosing what to Install .....	29
Figure 7.	Choosing the I/O Arrangement Manually .....	29
Figure 8.	Choosing the I/O Board Facility on the DPX-E120.....	30
Figure 9.	Pop-up Window for Installation Device-drivers.....	31
Figure 10.	Windows XP Installer Asking for Confirmation of Installation of Unsigned-driver .	31
Figure 11.	Windows Security Dialog on Windows 7 Driver Installation.....	32
Figure 12.	DirectPCI Product Remove Interface (“Uninstaller”).	34
Figure 13.	Linux Installation Types .....	35
Figure 14.	Debugging Word Bit-field Values .....	40
Figure 15.	DirectPCI API Library Debug Severity Values.....	41
Figure 16.	DirectPCI API Library Debugging Word Bit-field Values .....	42
Figure 17.	List of Demonstration Programs .....	45
Figure 18.	Compiling the Demonstration Code under Linux.....	45
Figure 19.	Device Manager Window Showing DirectPCI SRAM and ROM Devices .....	48
Figure 20.	Advanced Properties Tab for the DirectPCI SRAM and ROM devices .....	48
Figure 21.	Software Version Number Components.....	50
Figure 22.	Documtnation Version Number Components.....	51
Figure 23.	DirectPCI APIs Categorised by Gaming Function .....	53
Figure 24.	List of Devices Supporting Data Storage .....	54
Figure 25.	APIs for Handling Digital Inputs and Intrusion Circuits .....	56
Figure 26.	Description of All APIs .....	57
Figure 27.	Drivers, Headers and Supported APIs .....	59
Figure 28.	SRAM API and Driver Files .....	61
Figure 29.	Setting the UAC Execution Level for Programs Acccesing SRAM .....	62
Figure 30.	ROM API and Driver Files .....	81
Figure 31.	DPCI API and Core Driver Files.....	95
Figure 32.	Members of struct dpci_event.....	106
Figure 33.	Types of event for struct dpci_event.de_type.....	107
Figure 35.	Error Codes Common to All IDLP API Calls.....	123
Figure 37.	IDLP Firmware Sleep-time Settings .....	129
Figure 44.	Linux Native Input Device – struct input_event .....	178
Figure 45.	Event Codes numbers for DirectPCI Digital Inputs.....	179
Figure 61.	Distribution Share Menu .....	287
Figure 62.	DirectPCI folder .....	288
Figure 63.	\$OEM\$ Folder .....	289
Figure 64.	\$\$ Folder .....	289
Figure 65.	System32 folder .....	290
Figure 66.	Insert Oem Folder Path .....	290
Figure 67.	Explore Distribution Share Menu .....	291
Figure 68.	Out-of-Box Drivers Folder .....	292
Figure 69.	Distribution Share Pane Out-of-box Drivers .....	293
Figure 70.	Component Database Manager Window .....	293
Figure 71.	Importing Components into Microsoft Windows XP Embedded .....	294
Figure 72.	Output Window on Successful Component Import.....	295

Figure 73. Devices supported by the memutil command ..... 318

Page Intentionally Blank

# 1

## INTRODUCTION

### 1.1 OVERVIEW

The Advantech Innocore DPX-series computer range includes many features developed specifically for the gaming and embedded single-board computer industry including:

- Digital inputs and outputs
- battery-backed SRAM
- EEPROM storage
- chassis intrusion detection
- extra ROM sockets
- secure key storage, and SHA-1 computation through TCPA/TPM compliant hardware.
- Many other options for connection of I/O and storage peripherals

An API has been produced to ease the development, migration and to aid portability of software applications that require use of this functionality.

This manual documents the DirectPCI SDK & Run-time product and the facilities it provides:

- Support for both Windows and Linux
- Device drivers and programming API library in static and dynamic formats;
- Demonstration programs and Sample code
- Utility programs
- Components for Windows XP Embedded and Windows Embedded Standard 2009

- Support for Windows 7 and Windows 8 (32- and 64-bit) and support for Linux (32- and 64-bit distributions)
- Both release and debug quality binaries.
- Support for treating SRAM memory as a disk-drive in Windows or Linux.

The programming API provides functions pertaining to digital I/O, intrusion detection and logging, watchdog timers, SRAM and ROM, EEPROM storage and I<sup>2</sup>C busses.

This version of the manual corresponds to the API as implemented in the version 3.0 release and subsequent scheduled maintenance updates.

Across all operating systems, API entries conform to the same function name and parameters to ensure the user has the flexibility to select what operating system they would like to develop on with the option to migrate from one OS to another with the minimal effort.

Each API entry is listed by category. An example item is shown:

Figure 1. Example API entry

<b>Title</b> 3.2.9 <b>INTRUSION DETECTION FUNCTIONS: GET INTRUSION LINE STATUS</b>	<b>Header files required</b>  <b>API function prototype</b> <pre>#include &lt;dpci_core_api.h&gt; int dpci_id_intrusionstatus(void); #define INTRUSION0_MASK 0x01 #define INTRUSION1_MASK 0x02 #define INTRUSION2_MASK 0x04 #define INTRUSION3_MASK 0x08 #define INTRUSION4_MASK 0x10 #define INTRUSION5_MASK 0x20</pre> <b>Useful macros for use with this function</b>
	<b>Definitions</b> <pre>#include &lt;dpci_core_api.h&gt;</pre>
	<b>Parameters</b> <pre>None</pre>
	<b>Description</b> <p>This function returns a bit mask indicate the state of each intrusion line. A bit is set in the mask if the corresponding intrusion input is shorted to ground, i.e. closed circuit.</p>
	<b>Return Value</b> <pre>-1      The intrusion status could not be read &gt;=0    The intrusion status bit-mask.</pre>
	<b>Notes</b> <p>This function is only available on the DPX-116 and newer boards where the PIC firmware revision is 24 or greater. This function is not supported on the DPX-114 or DPX-115 boards.</p> <p>The <code>dpci_id_intrusionstatus()</code> function first appeared in the Direct PCI API version 1.2.1.</p>

## 1.2 SUPPORT

### 1.2.1 HARDWARE PLATFORMS

The following hardware boards have support in version 3.1:

- DPX-S435 (f/w revision 0Fh and newer) and compatible customer-specific variants;
- DPX-S410, S415, S420, S425, S430, E115, E130 and C710 (f/w revision 0x71 or newer);
- DPX-C705, C605, E105 and E120 (f/w revision 0x81 or newer);
- I/O board II 80-0062 (f/w revision 02 or newer).

### 1.2.2 HARDWARE PLATOFMRS No LONGER SUPPORTED

Official support for the following boards has been removed since version 3.0 as these boards are either no longer available or not recommended for new designs.

- DPX-116 and DPX-116U
- DPX-117 and DPX-112
- DPX-114 and DPX-115.
- I/O board 80-1003.

If the dpci\_core driver loads on a platform where the DirectPCI f/w revision does not meet the minimum required revision then the dpci\_core driver emits an error message to the system message log (see section 1.7.2 on debugging under Windows) but the driver will continue to operate as best it can.

### 1.2.3 OPERATING SYSTEMS: MICROSOFT WINDOWS

The following variants of Microsoft Windows are supported:

- Microsoft Windows XP Professional 32-bit SP2 and SP3
- Microsoft Windows XP Embedded 32-bit (SP2 or SP3, including FP2007)
- Microsoft Windows Embedded Standard 2009 32-bit
- Microsoft Windows 7 (with or without SP1) (32-bit and 64-bit)
- Microsoft Windows 7 (with or without SP1) Embedded (32-bit and 64-bit)
- Microsoft Windows 8/8.1 (32-bit and 64-bit)
- Microsoft Windows 8/8.1 Embedded (32-bit and 64-bit)

#### 1.2.3.1 *Operating Systems: Microsoft Windows 10*

It is anticipated the support for the forthcoming Microsoft Windows 10 operating system release will be fully validated in the first maintenance release (v3.1.1) of DirectPCI. At the time of writing, Windows 10 compatibility is considerable and no faults have been found that are specific to Windows 10.

## **1.2.4 OPERATING SYSTEMS: LINUX**

Almost all Linux variants can be supported however the following distributions are the key targets among Advantech Innocore's customer base and thus these are the ones that receive most test time.

- Ubuntu Linux 8.04 – 15.04 (Debian users should have no issues installing)
- Fedora Core 14 – 22

Slackware and Gentoo users or users of other distribution with hand-crafted kernels should have few problems but some reconfiguration might be required.

## **1.2.5 CUSTOMER SUPPORT**

If you have problems then you may e-mail Advantech Innocore for support at [support@innocoregaming.com](mailto:support@innocoregaming.com). Please send information regarding your problem to this e-mail address. Additional information which is often helpful to us includes:

- DirectPCI software driver and API versions (use the command *dpci -v*);
- operating system name and version;
- any log files you may have containing relevant information;
- the board type and revision
- BIOS revision.

## **1.2.6 THREAD-SAFETY**

All API functions are designed for thread safety. Most thread-safety is implemented in the kernel device-driver. Refer to specific functions or APIs for information about the specific implications of thread-safety features.

## **1.2.7 .NET SUPPORT**

A managed library has been produced to ease the development, migration and to aid portability of software applications that require use of the Microsoft .NET Framework.

This manual includes documentation of the managed DirectPCI API:

- Programming API library
- Demonstration program and Sample code

The programming API provides safe managed interface to the functions in the native DirectPCI API. This version of the manual corresponds to the API as implemented in the DirectPCI SDK & Run-time Programmers' Reference Manual version 3.1.0.

Most of the managed API entries conform to the same function names, parameters and return values as the native C-language API. This is to ensure the user has the flexibility to change the design platform from managed to native, or to select the host operating system they would like to develop on with the option to migrate from one OS to another with minimal effort.

In a few cases the managed API substitutes the native unsafe function prototypes with suitable managed and safe ones to enable the use of the managed DirectPCI library from any .NET language.

The managed APIs are contained in the namespace DirectPCI.

All custom types and constants used by the native API are re-declared in the managed library to ease the developer.

Usage example: *DirectPCI.DigitalIO.dpci\_io\_numports();*

Each API is implemented as a separate class. Each class is named after the corresponding API from the native DirectPCI library. All methods are static (VB Shared) and do not require instantiation of the class. The names of the methods correspond to the methods described in the DirectPCI SDK & Runtime Programmers Reference Manual v3.1.0.

The managed API is implemented in C++/CLI encapsulating all unsafe constructs.

An attempt has been made that all required marshaling between the native and managed worlds is implemented explicitly.

## **1.2.8 .NET DEMO PROGRAMS**

A demo program has been created to demonstrate the use of each API method. The demonstrator is implemented as a plug-in based framework where each DirectPCI API is demonstrated as a separate plug-in.

The demonstrator and source code are provided "as is". Comments and suggestion are welcome, however support for the demo program will not be available.

The demonstrator is accompanied by a .sln file to allow the project to be loaded into Microsoft Visual Studio 10 and later products.

### **1.2.8.1 Demo project structure**

DirectPCI.sln

- |\_ libDPCINET - the .NET API DirectPCI API library wrapper
- |\_ DirectPCINET - the demo application
- |\_ PluginInterface - the plug-in Framework
- |\_ Utilities - various utilities for the UI
  - | |\_ Input Data validator - provides methods for data validation(binary, hex, numeric)
  - | |\_ Vertical Progress Bar Control - provides vertical level meter
  - | |\_ Fixed Tab Control - provides Tab Control with vertical Tab labels
  - | |\_ Hex Viewer Control - provides a simple Hex viewer
  - | |\_ Split Button Control - provides multiple actions selection from a single button
  - |
- |\_ Plug-ins - All plug-ins as separate projects

### 1.2.8.2 Generic plug-ins project structure

Projects in the .NET demo programs have a number of files with the following common naming convention, where xyz represents name of an DirectPCI API.

xyzPlugin.csproj	plug-in project file
xyzPlugin.cs	plug-in interface
xyz.cs	demonstrates how to call the .NET DirectPCI API methods
xyzControl.cs	UI to the API methods

Please note: The demo program aims to demonstrate how to call the DirectPCI API methods in your application; it does not represent an actual gaming application.

## 1.3 FEEDBACK

We are always pleased to receive feedback on our products – whether positive or negative. You may always send feedback to our support e-mail address (see above).

## 1.4 CHANGES IN THE DIRECTPCI SDK & RUN-TIME V3.1

The key features and updates in the v3.1.1 release are:

- New API for host board identification (see 2.22 ).
- New API for managing debugging messages (see 2.21 ).

The key features and updates in the v3.1.0 release are:

- Support for the DPX-S435, DAC-BJ08 and DPX-E130 platforms.
- The installer and all .EXE and .DLL files are now signed on Windows 7/8 builds.
- Support for changing the drive letter for SRAM and ROM drives (see 1.10 ).
- Updated IDLP API (see 2.8 for capabilities introduction in IDLP f/w v60.xx).
- Usability improvements to demos and utilities.
- Improved IDPROM API (see 2.9 ): support for new API functions
  - *dpci\_idprom\_page\_status()*
  - *dpci\_idprom\_lock\_page ()*
  - *dpci\_idprom\_get\_copyprotect ()*
  - *dpci\_idprom\_set\_copyprotect()*
  - *dpci\_idprom\_write\_mfid ()*
  - *dpci\_idprom\_read\_mfid ()*
  - *dpci\_idprom\_write ()*
  - *dpci\_id\_readchecksum ()*

- *dpci\_id\_readevent\_instance()*
  - *dpci\_e130\_bps\_enable()*
  - *dpci\_e130\_bps\_is\_enabled()*
  - *dpci\_e130\_bps\_is\_charged()*
  - *dpci\_bios\_size()*
- For a list of changes in previous versions of the product, please refer to Appendix 1.
  - For a full list of other changes, including bugs fixed in subsequent maintenance releases, please refer to the CHANGE\_LOG.txt file included in all distribution archives.

#### 1.4.1 DEPRECATED FEATURES

On Linux, the old Linux-only event delivery method available via /dev/input/dpci is now deprecated. This facility will be removed in the next minor (v3.x) version of DirectPCI SDK & Run-time. As use of the facility does not require libdpci, programs that use it will continue to compile normally without warnings.

On both Windows and Linux, it is no longer possible directly to read from or to write to DirectPCI I/O register space unless debug level 0x8000 is set. There are no reasons why direct register access is required except for rare debugging cases.

(Previously, debug level 0x8000 only allowed access to key system registers.)

The following functions are now deprecated and may be removed in a future version:

Figure 2. API Entries deprecated in v3.1.0

Function Name	Suggested replacement
<a href="#"><u>dpci_io_read8()</u></a> (2.10.12)	DirectPCI digital input ports are best accessed using either:
<a href="#"><u>dpci_io_read16()</u></a> (2.10.13)	<ul style="list-style-type: none"> <li>• <a href="#"><u>The DirectPCI event-streaming API</u></a></li> </ul>
<a href="#"><u>dpci_io_read32()</u></a> (2.10.14)	<ul style="list-style-type: none"> <li>• <a href="#"><u>The DirectPCI Digital I/O API – read digital input port.</u></a></li> </ul> <p>Using these functions to access the registers of the DirectPCI I/O facility for any other purpose is no longer supported. Attempts to use these functions to control system level registers will be rejected by the device-driver.</p>

Function Name	Suggested replacement
<a href="#"><u>dpci_io_write8()</u></a> (2.10.15) <a href="#"><u>dpci_io_write16()</u></a> (2.10.16) <a href="#"><u>dpci_io_write32()</u></a> (2.10.17)	<p>DirectPCI digital output ports are best controlled using either of the functions</p> <ul style="list-style-type: none"> <li>• <a href="#"><u>DirectPCI Digital I/O API – write digital output port()</u></a></li> <li>• <a href="#"><u>DirectPCI Digital I/O API – change digital output port()</u></a></li> </ul> <p>It is no longer supported to use these functions to access the registers of the DirectPCI I/O facility for any purpose. Attempts to use these functions to control system level registers will be rejected by the device-driver.</p>
dpci_id_request_pending_event() dpci_id_cancel_pending_event()	<p>These functions were not documented and added only for specific customers using Microsoft Windows XP.</p> <p>They were marked as deprecated in v3.0 and in v3.1 they have been replaced with stubs which return an error. We recommend that customers using these instead use <a href="#"><u>The DirectPCI event-streaming API</u></a>.</p>
dpci_bat_read()	<a href="#"><u>dpci_bat_get_status()</u></a>

## **1.5 WINDOWS INSTALLATION**

The drivers are targeted at Microsoft™ Windows 7™, Microsoft™ Windows Embedded 7, Microsoft Windows 8, Microsoft Windows 8e, Microsoft™ Windows XP™, Microsoft™ Windows™ Embedded Standard 2009 and Microsoft™ Windows XP Embedded™ and all drivers are fully WDM-compliant.

All driver accessories (for example .INF files) have been developed to be compatible with these operating systems only.

All API and test applications are independent of versions of Windows and therefore only make standard Windows API calls used across the platform. No external (non-Windows) libraries have been used except those included in the Innocore board support package.

### **1.5.1 OBTAINING THE CORRECT INSTALLER FILES**

Unpack the .zip file and run the setup.exe program. The name of the zip file archive indicates the intended target platform.

<b>Downloaded Archive Name</b>	<b>Target OS</b>
dpci_install_3.1.yyy.xxx_release_win32_i386.zip	- for Microsoft Windows XP 32-bit only
dpci_install_3.1.yyy.xxx_release_win7_i686.zip	- for Microsoft Windows 7 32-bit only
dpci_install_3.1.yyy.xxx_release_win7_x64.zip	- for Microsoft Windows 7 64-bit only.

Figure 3. Windows Installation Types

It is advised to install only the correct build of software for the target OS you have. Installing any other version is supported but you will be unable to install the device drivers on an incompatible operating systems.

It is not possible to install Linux binaries on Windows systems or vice-versa for the purpose of supporting cross-compilation environments. To do this, you must install on a compatible OS (linux on linux or Windows on Windows and then manually copy the binaries required for your build environment.

### **1.5.2 RELEASE NOTES AND CHANGE-LOG**

The installation archive comes with a set of release notes and a change-log. Please read these files prior to installation for breaking news on changes and issues not covered in the current manual.

### **1.5.3 PREREQUISITES**

You should remove all previous versions of this product, but in particular the InstallShield-based versions v1.2.1 and v1.3.0. It is not enough to remove the installer folders themselves. Note, however, that the drivers need not be removed. If the uninstall option fails, you may manually remove the installation like this:

1. Use regedit to remove registry data found at the key HKLM\Software\Microsoft\Windows\CurrentVersion\Uninstall\{87E4F857-AA8A-4EAC-92D7-8805476781E0} and all its children.
2. Remove the folder C:\Program Files\Innocore Gaming Ltd\DirectPCI SDK & Run-time and all its children.

#### 1.5.4 RUNNING THE INSTALLER

The installer is an executable file called *setup.exe*. Installation proceeds via a guided user-interface. Note that on Windows 7 and later Windows operating systems, the *setup.exe* file is digitally signed and the publisher's name should show as "Advantech Co Ltd".

Figure 4. Splash-screen on running Setup.exe



#### 1.5.5 CHOOSING WHAT TO INSTALL

A range of different installation options is possible:

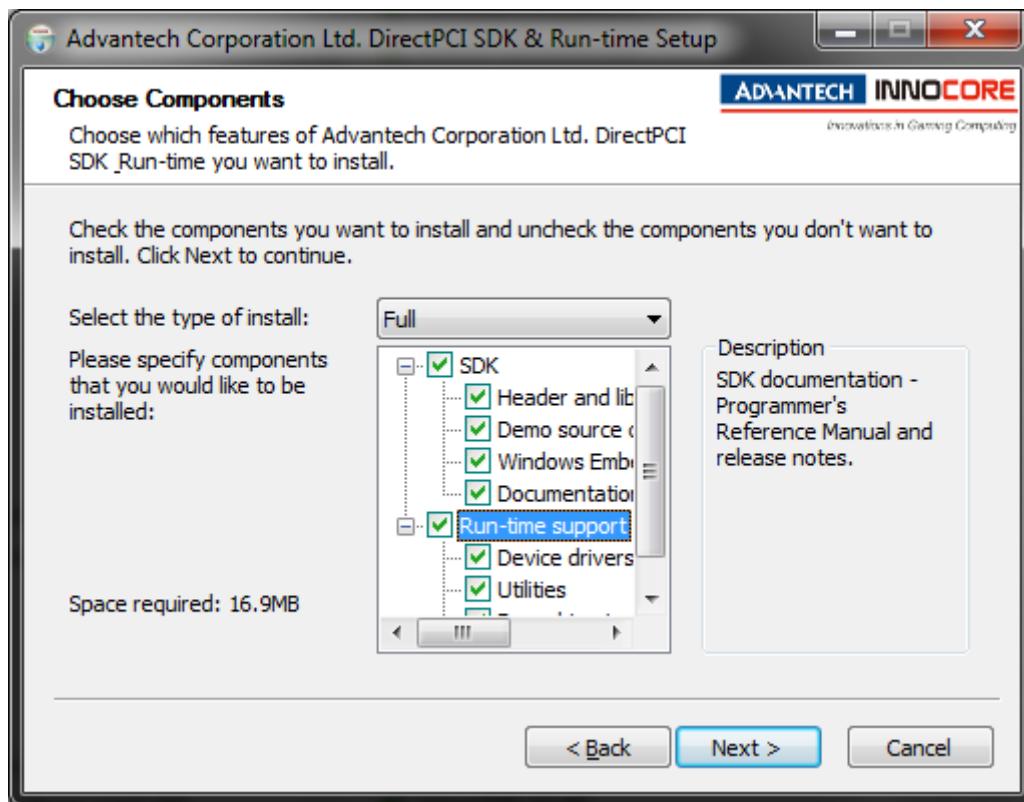
Figure 5. Types of Installation

Type of Installation	What You Get
<i>Run-time-only</i>	<ul style="list-style-type: none"><li>• Device drivers</li><li>• DLLs</li><li>• Utilities</li><li>• binaries for the demonstration programs</li></ul>

Type of Installation	What You Get
<i>SDK-only</i>	<ul style="list-style-type: none"> <li>• Header files;</li> <li>• libraries in production and debug quality builds;</li> <li>• libraries for both static and dynamic linking;</li> <li>• Components for Microsoft Windows Embedded Standard 7 (win7 versions)</li> <li>• Components (in .sld format) for Microsoft Windows XP target designer (win32_i386 versions)</li> <li>• Source code for the demonstration programs;</li> <li>• This manual and any other documentation.</li> </ul>
<i>Full</i>	Everything from Run-time-only and SDK-only together.
<i>Custom</i>	Your own chosen combination of components.

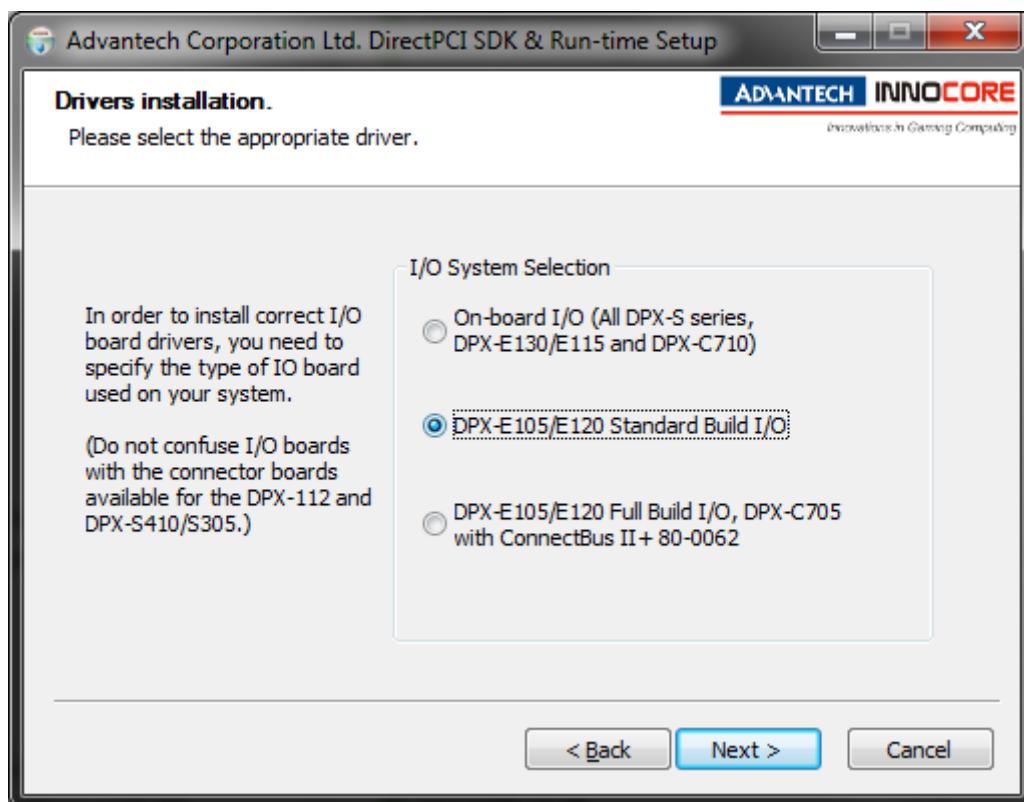
An additional screen (below) shows the available installation options:

Figure 6. Window for Choosing what to Install



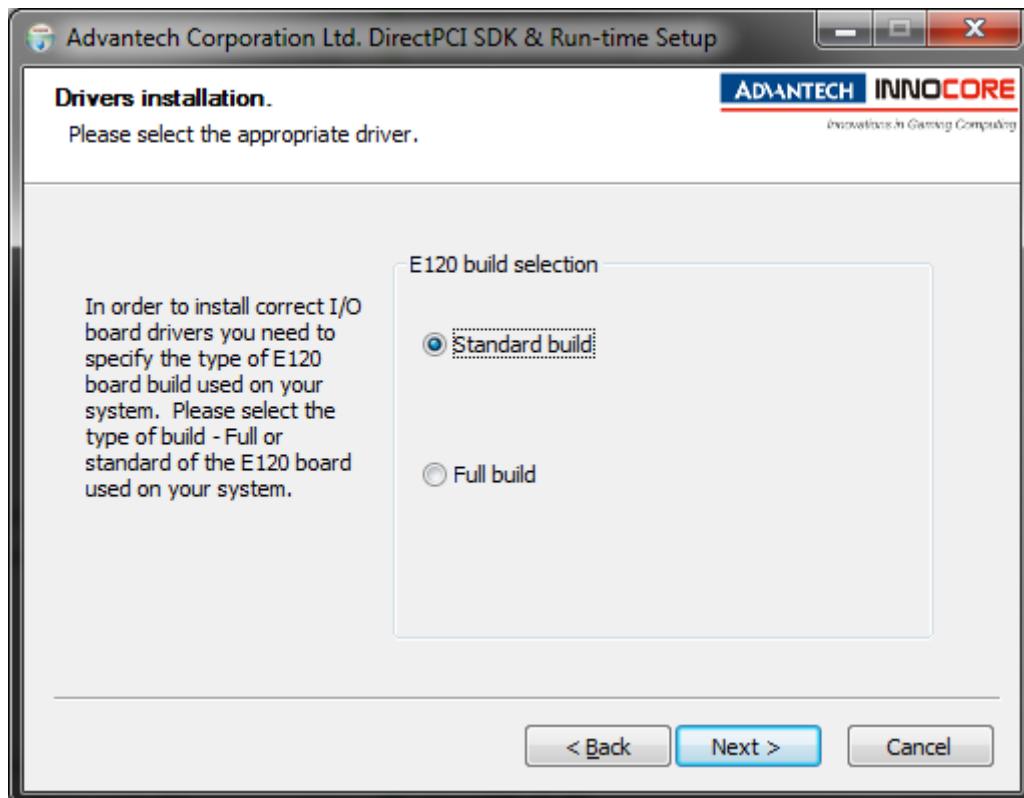
If you run the setup.exe file on a DPX-series board then ordinarily, the installer will successfully detect which board it is and correctly install the correct device driver for the type of I/O you have. However, if you run the installation on a non-Innocore, or, if the setup.exe file is unable to determine which board you have, then you will be asked to state what kind of I/O arrangement is present:

Figure 7. Choosing the I/O Arrangement Manually



You should select the correct type of board as it may be difficult to correct this choice later.

Figure 8. Choosing the I/O Board Facility on the DPX-E120

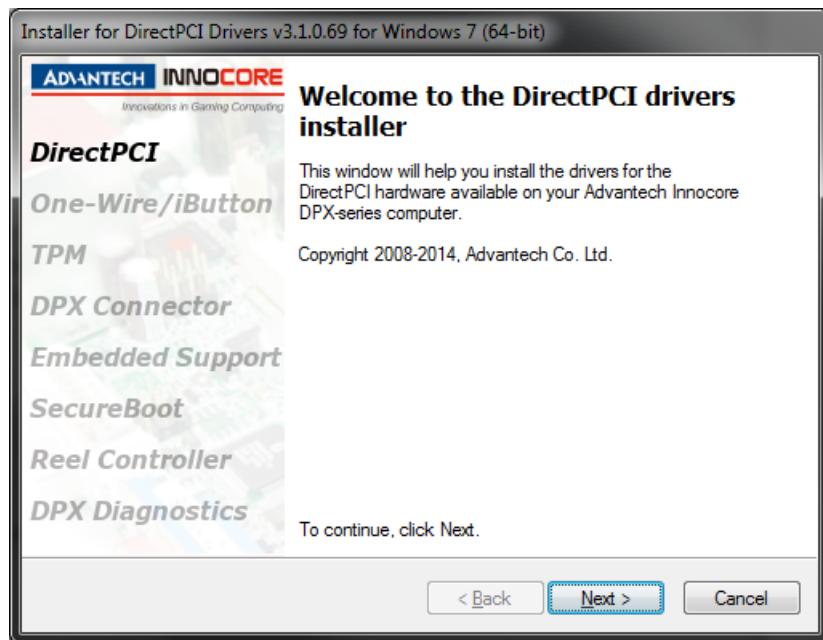


Note that the *Standard Build DPX-E120* and *DPX-E105* is the most commonly available build. This build supports only 6 COM ports. The full build is a special build of the product with 4 additional COM ports (providing a total of 10).

### 1.5.6 INSTALLING THE RUN-TIME DEVICE-DRIVERS AND LIBRARIES

If you choose to install the run-time device-drivers, then an additional window will appear to complete their installation. For example:

Figure 9. Pop-up Window for Installation Device-drivers



When installing the run-time drivers on Microsoft Windows XP, you will be asked whether to accept unsigned drivers.

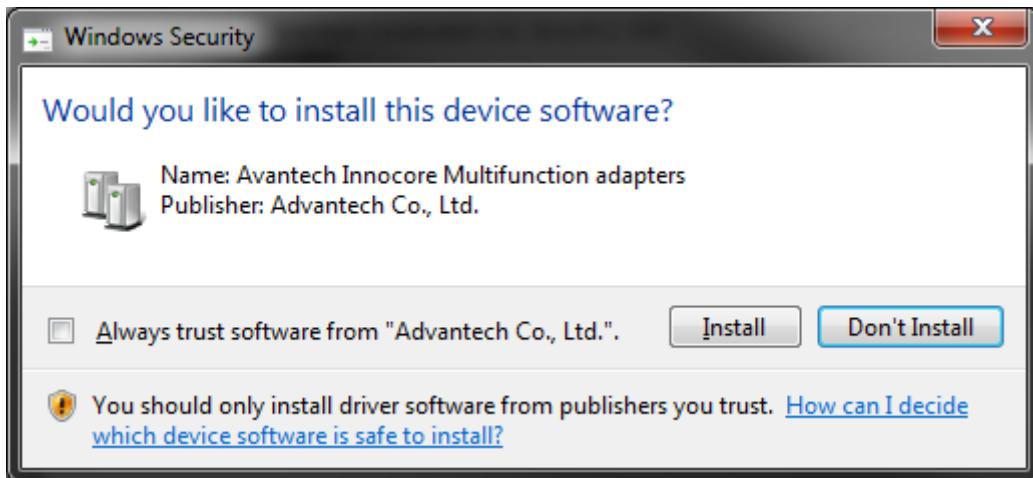
Figure 10. Windows XP Installer Asking for Confirmation of Installation of Unsigned-driver



You should click Continue Anyway as Innocore device-drivers are never signed on Windows XP.

On Windows 7, Windows 8 and Windows 10, a dialog box such as that below will appear. You should choose the lower option (*Install this driver software anyway*) in order to complete.

Figure 11. Windows Security Dialog on Windows 7 Driver Installation



You are not required to choose the “Always trust software from Advantech” button to successfully install the drivers.

### **1.5.7 SHARED AND STATIC LIBRARIES**

The DirectPCI library has been provided in two forms: a stub library *libdpci.lib* for use with the DLL, and a static library *libdpci\_static.lib* for use where the DLL is not desired.

### **1.5.8 CHANGING FROM 80-1003 TO 80-0062 I/O BOARDS ON MICROSOFT WINDOWS SYSTEMS**

If you wish to make this change then you must proceed carefully: you must launch Device Manager, go to the ports section and manually uninstall each COM port first. Ensure that you do not invoke a scan for hardware changes action either by rebooting before you have removed the hardware or by using the option of the same name in Device manager. You may then physically remove the DirectPCI hardware. After that, you may re-install the drivers.

### **1.5.9 SRAM/ROM DRIVER INSTALLATION NOTE**

The SRAM function driver does not always install correctly the first time because a Microsoft®-supplied signed driver is installed by default. To install the SRAM driver, open Device Manager (Control Panel, System window, Hardware tab then Device Manager button) and look for the item PCI Standard RAM Controller in the System Devices area. Now right-click on that item and select Update driver and following the dialog through, always choosing the options to allow you to choose manually the appropriate driver.

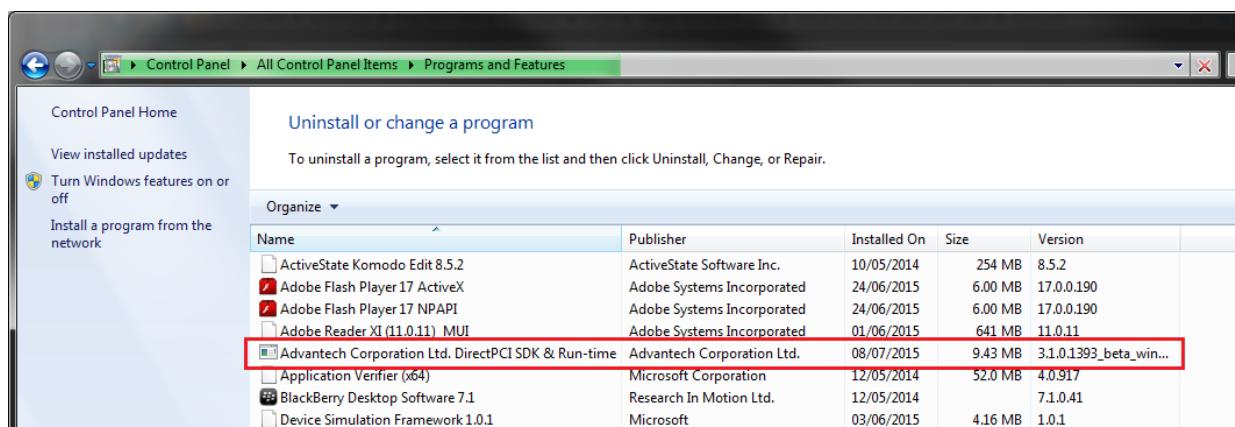
## 1.5.10 SRAM & ROM DEVICES IN DISK-DRIVE (BLOCK) MODE

Starting from version 2.2.0, the SRAM and ROM are available as block devices. The SRAM is accessible as drive letter B. It can be formatted to host a FAT file system. The ROM is accessible as drive letter R. A valid FAT format file-system image on the ROM can be accessed via the ROM-drive.

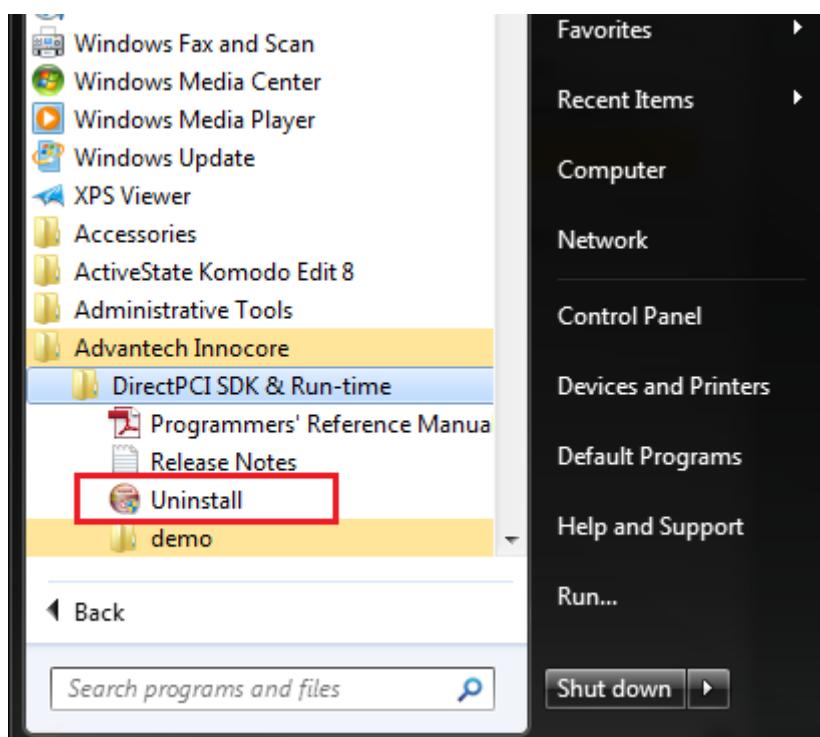
## 1.5.11 REMOVING THE PRODUCT

It is possible to remove the product by running its *Uninstall.exe* program. This can be invoked in one of three ways:

- Via the Control Panel ‘Programs and Features’ window:



- Via the Start menu



From the command line by changing directory to the installation folder and then manually running *uninstall.exe*.

```
C:\Program Files\Advantech Innocore\DirectPCI SDK & Run-time>dir
Volume in drive C is Windows 7 Ent SP1
Volume Serial Number is 86C9-04A1

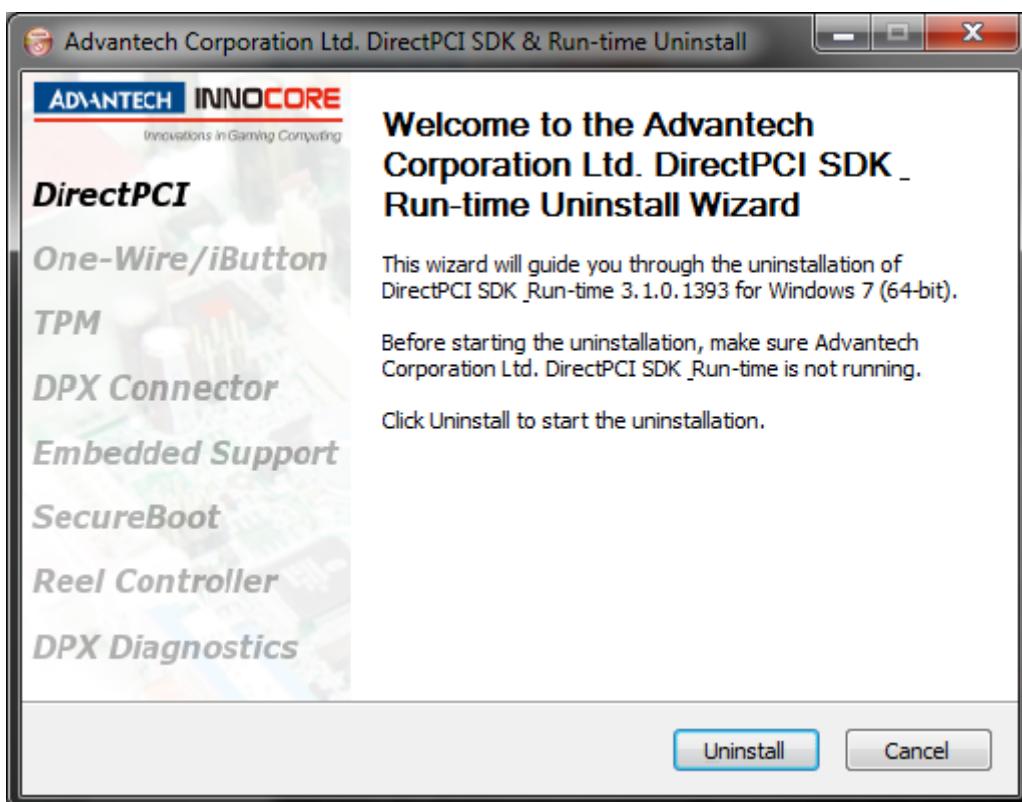
Directory of C:\Program Files\Advantech Innocore\DirectPCI SDK & Run-time

08/07/2015  16:46    <DIR>          .
08/07/2015  16:46    <DIR>          ..
08/07/2015  16:46    <DIR>          demo
08/07/2015  16:46    <DIR>          Documentation
08/07/2015  16:46    <DIR>          driver
08/07/2015  16:46    <DIR>          embedded
08/07/2015  16:46    <DIR>          include
08/07/2015  11:31      67,692  Innocore.ico
08/07/2015  16:46    <DIR>          lib
08/07/2015  16:46    <DIR>          lib.DEBUG
08/07/2015  16:46      148,794  Uninstall.exe
08/07/2015  16:46    <DIR>          Utilities
               2 File(s)     216,486 bytes
              10 Dir(s)   340,877,819,904 bytes free

C:\Program Files\Advantech Innocore\DirectPCI SDK & Run-time>
```

In all cases, the uninstaller presents this interface:

Figure 12. DirectPCI Product Remove Interface ("Uninstaller")



We regret that it is not possible at present to remove individual components of the product. Note also that the *uninstall.exe* program is not digitally signed as the *setup.exe* is.

## 1.6 LINUX INSTALLATION

### 1.6.1 OBTAINING THE CORRECT INSTALLER FILES

Unpack the archive file and run the *install* binary program. The name of the archive file archive indicates the intended target platform.

Downloaded Archive Name	Target Hardware + OS
dpci_install_3.0.yyy.xxx_release_linux_i686.tar.bz2	- for Linux on x86, 32-bit only
dpci_install_3.0.yyy.xxx_release_linux_x86_64.tar.bz2	- for Linux on x86_64, 64-bit only.

Figure 13. Linux Installation Types

It is advised to install only the correct build of software for the type of OS you have. Installing any other version is not supported and will ordinarily be refused or yield undefined results.

The installer for the 64-bit product is a 64-bit executable itself and will not execute on a 32-bit installation even if the underlying hardware supports 64-bit execution.

On a 64-bit system, although it may be possible to execute the *install* program from the 32-bit distribution, note however that the DirectPCI API library will be a 32-bit version as will be the utility programs. The demo programs and kernel drivers should also compile and work correctly; however, these will automatically be 64-bit binaries and thus unable to use the 32-bit library.

### 1.6.2 RELEASE NOTES AND CHANGE-LOG

The installation comes with a set of release notes and a change-log. Please read these files prior to installation for breaking news on changes and issues not covered in the current manual.

### 1.6.3 PREREQUISITES

The following are pre-requisites to installation on ALL distributions:

- The Linux kernel headers and module build files for a kernel v2.6.5 or newer. The directory `/lib/modules/<kernelver>` must exist and contains valid links called "build". Kernels v3.x are now supported too.
- The GNU C compiler (gcc), v3.3.3 or newer
- GNU Libc v2.6 (32-bit builds) or v2.11 (64-bit builds)
- binutils (ld, as etc.)
- GNU make
- setserial (if you have an 80-0062 I/O board).

If you have purchased a development kit from Advantech Innocore then all the packages required for installation are already installed.

## **1.6.4        INSTALLATION**

This installation process should be run ideally without the existing modules loaded into memory. You should close all applications before installing.

The headers for the kernel you are running must be installed and correctly linked from the modules directory /lib/modules/<kernel-ver> with a `build' link to an appropriate place. Without these in place, the installation will fail.

To install the software, log in as root and run the install script.

```
$ su -  
Password:xxxx  
# tar xfj dpci_install_3.0.yyy.xxx.tar.bz2  
# cd dpci_install_3.0.yyy.xxx  
# ls  
total 248  
-rw-r--r-- 1 aidan users 3515 2009-11-22 11:40 CHANGE_LOG.txt  
-rw-r--r-- 1 aidan users 10027 2009-11-22 11:40 RELEASE_NOTES.txt  
-rwxr-xr-x 1 aidan users 235552 2009-11-22 11:40 install  
# ./install
```

You will then be guided through the simple installation process.

The drivers and demos are distributed in source code form under the GPL license. Please note this does not change your obligations under the NDA your company has with Advantech Innocore.

No API source code is available. Compiled drivers are installed in the appropriate place under /lib/modules. Header files are placed in /usr/include and API library files in /usr/lib. Demonstration source code is installed in /usr/src/InnocoreGaming.

## **1.6.5        UNATTENDED INSTALLATION**

There are two modes of unattended installation possible under Linux.

- To install as normal but without the user interface and confirmation prompts, issue the ./install –q command.
- To merely unpack the files provided by the installation into the current directory, issue the ./install –x command.

## **1.6.6        LINUX KERNEL UPDATES**

When using Linux distribution such as Ubuntu, Fedora Core, Debian and many others, downloads are made available with many bug fixes. When such an update causes a new kernel image to be installed then it will be necessary to re-run the build of the kernel drivers.

For example:

```
# cd /usr/src/InnocoreGaming/DirectPCI-3.0.1.xxx  
# make install-drivers
```

This will rebuild the drivers and install the binaries in the correct place. You will now need to reboot the system to load the new drivers.

It is also possible simply to rerun the installer binary provided when first using the drivers.

## **1.6.7        LINUX BOOTING**

The installer attempts to copy the script `dpci_drv.sh` to the correct place automatically so that DPCI drivers are correctly loaded each time the system boots. However, on some older distributions this is not so straight-forward.

If the directory `/etc/init.d` or `/etc/rc.d/init.d` exists, then the file is copied there. Then if `/etc/rc3.d` or `/etc/rc.d/rc3.d` exists then a symbolic link is made from the `init.d` directory.

For users using the original sysvinit scripts (e.g. Slackware users) or custom boot scripts, you may have to re-work the installation of the script. In such circumstances it is copied to the same directory as `rc.local` is found (either in `/etc` or `/etc/rc.d`). However, this may mean it is not started early enough in the boot sequence.

## **1.6.8        UDEV SUPPORT**

LDM (linux device model) and sysfs are both supported (since v1.2.0). This means that for systems with hot-plug support, modules will be loaded automatically and the appropriate nodes in `/dev` will also be created correctly whenever the system boots.

## **1.6.9        LINUX SERIAL PORTS**

### **1.6.9.1      *dpci\_serial Driver Applicability***

The DirectPCI run-time component includes the driver `dpci_serial`. This driver provides serial port support for the following systems that use the 80-1003 I/O board, the 80-0062 I/O board or the I/O support board:

- DPX-C705/C605
- DPX-E105
- DPX-E120

The `dpci_serial` driver does not work on any other systems, e.g. the DPX-S-series and other DPX-E and DPX-C series boards. The `install` program will fail to install it on these systems.

### **1.6.9.2      *Kernel Configuration Requirements***

Because the serial I/O interrupt is shared with the other DirectPCI interrupt sources, you must either enable shared interrupts in your kernel configuration (set `CONFIG_SERIAL_8250_SHARE_IRQ=y`) or add "`options 8250 share_irqs=1`" in `/etc/modprobe.conf`.

### **1.6.9.3      *Boot-time Serial Port Configuration***

With a normal vanilla kernel from [ftp.kernel.org](http://ftp.kernel.org), the `dpci_serial` module installs and provides serial ports `/dev/ttys4` - `/dev/ttys7`. If you use some vendors' kernels, (notably RedHat, Fedora and Debian/Ubuntu) then the port numbers may appear different (typically ports 14, 15, 44 and

45 are used instead if a large number of legacy serial ports is enabled using CONFIG\_SERIAL\_8250\_MANY\_PORTS = y). Ordinarily the dpci\_drv.sh script handles the numbering correction automatically at boot time.

On other distributions, the number of available serial ports configured at boot-time is limited to 2 or 4. In many cases this will cause the install to fail when it attempts to load the dpci\_serial driver. This can be fixed by ensuring the 8250 serial driver is configured for the correct number of required UARTs. This can be in two ways: when the 8250 driver is built into the kernel, add the configuration option 8250.nr\_uarts=12. When the 8250 driver is a module then add 8250.nr\_uarts=12 to /etc/modules.conf.

Alternatively run these commands before loading the dpci\_serial module manually:

```
# setserial /dev/ttys4 port 0 irq 0
# setserial /dev/ttys5 port 0 irq 0
# setserial /dev/ttys6 port 0 irq 0
# setserial /dev/ttys7 port 0 irq 0
# setserial /dev/ttys14 port 0 irq 0
# setserial /dev/ttys15 port 0 irq 0
# setserial /dev/ttys44 port 0 irq 0
# setserial /dev/ttys45 port 0 irq 0
# modprobe dpci_serial
```

When configuring the Linux kernel, it is advisable to set the configuration setting CONFIG\_SERIAL\_8250\_MANY\_PORTS=n (i.e. off) so that excessive support for legacy serial cards is not provided.

### **1.6.10 SHARED AND STATIC LIBRARIES**

The DirectPCI library has been provided in two forms: a shared library *libdpci.so* and a static library *libdpci.a* for use where the shared library is not desired.

## 1.7 DEBUGGING

We provide support to help you debug issues you experience during development, either with our library code or with your application:

- The SDK includes both debug versions of our drivers and libraries.
- We allow debug messages to be enabled selectively via the API or from the environment.
- Debug messages may be redirected to a file and or sent to an application-specification function.

### 1.7.1 DEBUG AND RELEASE BINARIES

This distribution contains debug and release versions of the API libraries. The difference between the two is:

- RELEASE: compiled with optimisation
- DEBUG: compiled without optimisation, debug symbols enabled and other symbols not stripped and some printf messages in key functions.

On Windows, the debug files can be found in the files\_debug sub-folder of embedded folder under the installation directory when the Windows XP Embedded Components part of the SDK is installed.

On Linux, the debug version of *libdpci* can be found in /usr/lib along side the regular release build files.

### 1.7.2 WINDOWS

It is not possible to recompile the drivers; however, a debug set of drivers is included. Ordinarily one can change between the release and debug drivers easily:

- Open up windows Device Manager
- Find the Core I/O, SRAM or ROM device to have its driver changed
- Right-click on the device and hit *Disable*
- Move the relevant .sys file for the driver into C:\WINDOWS\SYSTEM32\DRIVERS
- Right-click again on the device and now hit *Enable*.

To obtain the driver's debug output, you should visit <http://www.sysinternals.com/> and download the dbgview.exe program.

The debug levels are listed further below in section 1.7.4 (p40).

### 1.7.3 LINUX

You must compile the DEBUG drivers yourself. The source code is installed ordinarily under /usr/src/InnocoreGaming/<Release-name>. You should run "make" under the top-level directory.

For example:

```
# cd /usr/src/InnocoreGaming/DirectPCI-3.1.1.xxx
# make install-drivers
```

This process automatically builds both DEBUG and RELEASE quality drivers. By default release drivers are installed.

You may then change the debug level in two ways. When loading the driver with *insmod* or *modprobe*, add *debug=N* to the command line where N is the desired debug level; you may also use the command *dpci* and the -D argument. The source code file *dpci\_core\_priv.h* is in the driver build area and at the end of it is a map of the bits used in the debug word.

All appropriate driver diagnostic information is output using *printk()* calls for driver related functions and should thus appear in the system logs.

Debug output can then be viewed using:

- the *dmesg* command, which has a limited size buffer determined by the kernel configuration; or,
- */var/log/kernel* (typically) and or */var/log/messages* on most Linux distributions.

#### 1.7.4 CORE I/O DRIVER DEBUG LEVELS

The following table shows the debug levels for the Core I/O driver (*dpci\_core*).

Figure 14. Debugging Word Bit-field Values

Value	Type of output
!= 0	Any non-zero debug setting causes general debug messages to be emitted.
0x0001	Emit debug information about ioctl commands (linux) or device control commands (Windows)
0x0002	Emit debug information about register access.
0x0004	Emit debug information about IDLP access
0x0008	Emit debug information about access to register 0
0x0010	Unused.
0x0020	Emit debug information about digital I/O operations
0x0040	Emit debug information about I2C port operations
0x0080	Emit debug information about One-wire/iButton access
0x0100	Emit debug information about IRQ controls
0x0200	Emit debug information about Windows DDK functions (Windows drivers only).

Value	Type of output
0x8000	<p>This level does not cause output to be emitted. Instead, it allows programs to use the now-deprecated <code>dpci_core_write{8,16,32}()</code> and <code>dpci_io_read{8,16,32}()</code> functions allowing direct access to DirectPCI I/O register space.</p> <p>It is not recommended that this option be used except for rare debugging cases.</p> <p>This option may be removed in a future release.</p>

For the `dpci_mem` driver, the debug level is only ever on (1) or off (0).

## 1.7.5 DIRECTPCI API LIBRARY DEBUG LEVELS

The following tables shows the debug levels for use with the debug build of the DirectPCI library.

You can use enable debugging using two methods:

- By setting the variable `DPCI_DEBUG_LEVEL` in the cmd.exe window or in the global or local user environment. In this mode the value assigned to `DPCI_DEBUG_LEVEL` should be a number in decimal, octal (leading 0 required) or hexadecimal (leading 0x required). A quick way to enable all debug output is to assign the value -4 (equivalent to 0xffffffffc).
- You can also use the functions `dpci_api_set_debug_level()` (see 2.21.3) or `dpci_api_modify_debug_level()` (see 2.21.4) as provided by the Debug API.

By default, debug output is sent to the command window where the program was started. However, debug output may instead be directed to a file using two methods:

- By setting the variable `DPCI_DEBUG_LOGFILE` in the cmd.exe window or in the global or local user environment. The value assigned should be the fully qualified path of the file. If the path begins with a '+' then the '+' character is skipped and the file is opened so that new messages are appended to the end of any existing file. Otherwise, the log file will be truncated and new messages will always overwrite old ones.
- You can also use the functions `dpci_api_set_debug_output_file()` (see 2.21.7) or `dpci_api_set_debug_output_handle()` (see 2.21.8) as provided by the Debug API.

The following values relate to the lowest bits of the debug word and determine what type of messages are emitted.

Figure 15. DirectPCI API Library Debug Severity Values

Macro	Value	Description
<code>DPCI_DEBUG_SEVERITY_MASK</code>	0x0003	This bit-mask determines that the lowest 2 bits in the debug word are used determine the level of messages that are output.
<code>DPCI_DEBUG_SEVERITY_TRACE</code>	0x0000	<p>When processing a debug message, this level determines that the message is a marker for entry or exit from a particular function.</p> <p>When setting the debug-level, this value determines that all types of message are output.</p>

Macro	Value	Description
DPCI_DEBUG_SEVERITY_INFO	0x0001	<p>When processing a debug message, this level determines that the message is a supplementary message from a particular function.</p> <p>When setting the debug-level, this value determines that only this type of message, warning messages or error messages are output.</p>
DPCI_DEBUG_SEVERITY_WARNING	0x0002	<p>When processing a debug message, this level determines that the message is a warning message from a particular function.</p> <p>When setting the debug-level, this value determines that only this type of message or error messages are output.</p>
DPCI_DEBUG_SEVERITY_ERROR	0x0003	<p>When processing a debug message, this level determines that the message is an error message from a particular function.</p> <p>When setting the debug-level, this value determines that only this type of message is output.</p>

The following values refer to all but the lowest bits of the debug word.

Figure 16. DirectPCI API Library Debugging Word Bit-field Values

Macro	Value	Type of output
DPCI_DEBUG_API_MASK	0xffffffffc	Any non-zero debug setting causes general debug messages to be emitted
DPCI_DEBUG_CORE_INIT_HANDLE	0x00000004	Messages relating to opening the DPCI_CORE device
DPCI_DEBUG_IDLP_API	0x00000008	Messages relating to the IDLP functions
DPCI_DEBUG_TS_API	0x00000010	Messages relating to 80/0062 temperature sensor functions
DPCI_DEBUG_I2C_API	0x00000020	Messages relating to I2C functions
DPCI_DEBUG_EEPROM_API	0x00000040	Messages relating to EEPROM functions
DPCI_DEBUG_EVENT_API	0x00000080	Messages relating to Events and event handling functions
DPCI_DEBUG_EVENT_DEBOUNCE	0x00000100	Messages relating to digital input debouncing functions
DPCI_DEBUG_BIOS_API	0x00000200	Messages relating to BIOS functions
DPCI_DEBUG_BAT_API	0x00000400	Messages relating to battery functions
DPCI_DEBUG_IDPROMIDENT_API	0x00000800	Messages relating to the IDPROM
DPCI_DEBUG_DIGINOUT_API	0x00001000	Messages relating to digital inputs and outputs
DPCI_DEBUG_QMGPIO_API	0x00002000	Messages relating to the Quiet Mode API and the general purpose I/O pin API

<b>Macro</b>	<b>Value</b>	<b>Type of output</b>
DPCI_DEBUG_MISC_API	0x00004000	Messages not otherwise classified
DPCI_DEBUG_BOARDS_API	0x00008000	Messages relating to specific boards, e.g. the DPX-E130 API
DPCI_DEBUG_WD_API	0x00010000	Messages relating to the watch-dogs
DPCI_DEBUG_PUC_API	0x00020000	Messages relating to the PuC API
DPCI_DEBUG_PUC_COMMs	0x00040000	Messages relating to PuC communications
DPCI_DEBUG_SRAM_INIT_HANDLE	0x04000000	Messages relating to initialisation of the SRAM device
DPCI_DEBUG_SRAM_API	0x08000000	Messages relating to the SRAM API
DPCI_DEBUG_SRAM_MMAP	0x10000000	Messages relating to memory mapping the SRAM device
DPCI_DEBUG_ROM_INIT_HANDLE	0x20000000	Messages relating to initialisation of the ROM device
DPCI_DEBUG_ROM_API	0x40000000	Messages relating to the ROM API
DPCI_DEBUG_ROM_MMAP	0x80000000	Messages relating to memory mapping the ROM device

For the dpci\_mem driver, the debug level is only ever on (1) or off (0).

## **1.8 UTILITIES**

The following utility programs have been included to enhance the debugging and scripting experience. They are described in full in section 4.

batt	Returns the state of the batteries. (4.1 - p305)
dipsw	Returns the state of the four dip switches. (4.2 - p307)
dpci	Get a wide range of information about DirectPCI system. (4.3 - p308)
dpxname	States what kind of board the host is. (4.4 - p311)
idutil	Performs various functions on the IDLP. (4.5 - p312)
ioboard	For boards that support I/O expansion via ConnectBUS II+, this utility states what kind/revision of ConnectBUS II+ I/O board is installed. (4.6 - p315)
memutil	Provides access to SRAM, ROM & EEPROMs. (4.7 - p317)

## 1.9 DEMO PROGRAMS AND CODE SAMPLES

Almost all API function descriptions have been complemented by example code. The example code fragments show small pieces of code to demonstrate the anticipated use of the API functions. However, the samples are not complete functions or programs and so alone will ordinarily not compile alone into a working program. Each example section instead refers to a fully functional demonstration program – supplied as both executable binary and C-language source code - where the API functions are used.

The demonstration programs in the *demo* sub-directory of your Innocore DirectPCI SDK and Run-time installation has fuller example code as explained here.

Figure 17. List of Demonstration Programs

- batt\_demo      Demonstrates accessing batteries capabilities.
- bios\_demo      Demonstrates obtaining the BIOS image size and image data and saving it to a file.
- callback\_demo      Demonstrates using callbacks and de-bouncing
- eeprom\_demo      Demonstrates accessing EEPROM capabilities.
- i2c\_demo      Demonstrates accessing i2c busses
- idlp\_demo      Demonstrates accessing intrusion detection processor
- idprom\_demo      Demonstrates accessing DS2431 IDPROM capabilities.
- io\_demo      Demonstrates accessing DirectPCI I/O capabilities.
- rom\_demo      Demonstrates accessing ROM capabilities.
- sram\_demo      Demonstrates accessing SRAM capabilities.
- ts\_demo      Demonstrates accessing I/O board II temperature sensors
- uart\_demo      Demonstrates accessing COM ports. This is the only demonstration program that is different between Linux and Windows.

### 1.9.1 DEMONSTRATION CODE UNDER LINUX

Demonstration programs are provided in */usr/src/InnocoreGaming/DirectPCI-3.0.0.xx* on Linux platforms. Use the *make* command to build the examples:

Figure 18. Compiling the Demonstration Code under Linux

```
# cd /usr/src/InnocoreGaming/DirectPCI-3.0.0.300_release_linux_i686/demo
# make
+demo
*****
Building batt_demo
```

```
*****
Compiling /usr/src/InnocoreGaming/DirectPCI-
3.0.0.361_beta_linux_i686/demo/batt_demo.c (debug)
Linking executable DEBUG_linux_i686/batt_demo

*****
Building idlp_demo
*****
Compiling /usr/src/InnocoreGaming/DirectPCI-
3.0.0.361_beta_linux_i686/demo/idlp_demo.c (debug)
Linking executable DEBUG_linux_i686/idlp_demo

*****
Building io_demo
*****
Compiling /usr/src/InnocoreGaming/DirectPCI-
3.0.0.361_beta_linux_i686/demo/io_demo.c (debug)
Linking executable DEBUG_linux_i686/io_demo

*****
Building i2c_demo
*****
Compiling /usr/src/InnocoreGaming/DirectPCI-
3.0.0.361_beta_linux_i686/demo/i2c_demo.c (debug)
Linking executable DEBUG_linux_i686/i2c_demo

*****
Building eeprom_demo
*****
Compiling /usr/src/InnocoreGaming/DirectPCI-
3.0.0.361_beta_linux_i686/demo/eeprom_demo.c (debug)
Linking executable DEBUG_linux_i686/eeprom_demo

*****
Building ts_demo
*****
Compiling /usr/src/InnocoreGaming/DirectPCI-
3.0.0.361_beta_linux_i686/demo/ts_demo.c (debug)
Linking executable DEBUG_linux_i686/ts_demo

*****
Building idlp_threads
*****
Compiling idlp_threads.c (debug)
Linking executable DEBUG_linux_i686/idlp_threads

*****
Building callback_demo
*****
Compiling /usr/src/InnocoreGaming/DirectPCI-
3.0.0.361_beta_linux_i686/demo/callback_demo.c (debug)
Linking executable DEBUG_linux_i686/callback_demo

*****
Building sram_demo
*****
Compiling /usr/src/InnocoreGaming/DirectPCI-
3.0.0.361_beta_linux_i686/demo/sram_demo.c (debug)
Linking executable DEBUG_linux_i686/sram_demo

*****
Building rom_demo
*****
Compiling /usr/src/InnocoreGaming/DirectPCI-
3.0.0.361_beta_linux_i686/demo/rom_demo.c (debug)
Linking executable DEBUG_linux_i686/rom_demo
```

```
Building uart_test
*****
Compiling uart_test.c (debug)
Linking executable DEBUG_linux_i686/uart_test
-dpci_serial
-demo
#
```

Please note the following:

- when building the demos, the binaries are produced in the *DEBUG\_linux\_i686* subdirectory (32-bit installations) or the *DEBUG\_linux\_x86\_64* subdirectory (64-bit installations);
- the build system uses the *make* facility with custom rules; therefore it is only possible to perform a build in the root directory of the installation. The *make* program will fail if you attempt to change directory into one of the installation's sub-directories and run it there.

### **1.9.2 DEMONSTRATION CODE UNDER MICROSOFT® WINDOWS® XP**

Under Windows, the code samples (identical to the Linux ones) are installed normally under *c:\Program Files\Advantech Innocore\DirectPCI SDK & Run-time\demo*. Each demonstrator is accompanied by a *.dsp* file to allow the project to be loaded into one of the range of Microsoft Visual Studio products. The projects were created using Microsoft Visual C++ 6.0 and so it should be easy to import them into all modern versions of Microsoft Visual Studio.

## 1.10 CHANGING THE DRIVE LETTERS FOR SRAM AND ROM DRIVES

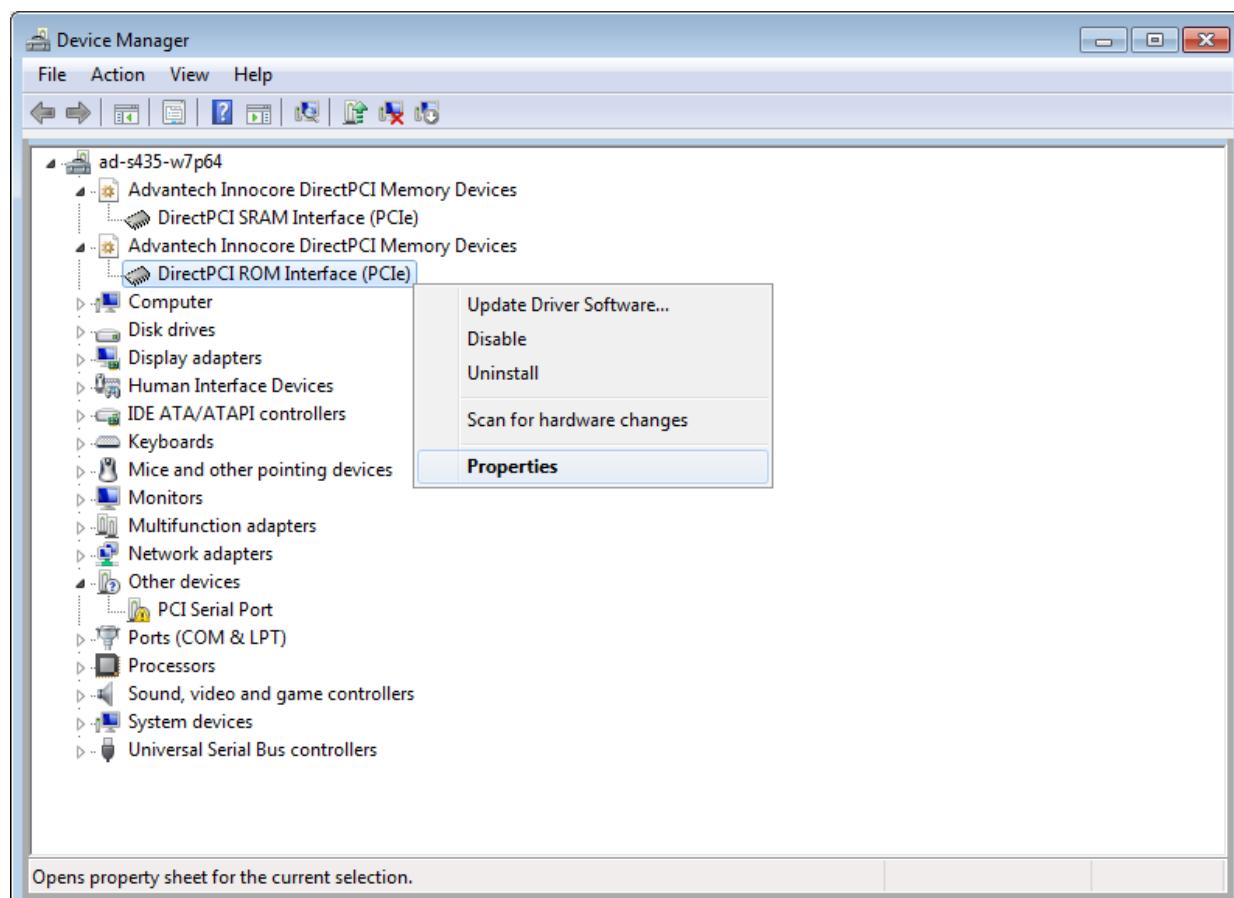
A new feature in version 3.1.x of DirectPCI SDK & Run-time is the facility to change the drive letter assigned to the SRAM and ROM block devices. Previously these were unchangeable: the SRAM was always drive B: and the ROM always drive R:.

In the updated implementation, the default drive letters remain the same as the old fixed drive letters. However, the SRAM drive default to being enabled and the ROM drive defaults to being disabled. Note that whether drive mounting is enabled or disabled, it is still possible to access the devices through the regular programming APIs.

### 1.10.1 PROCEDURE FOR CHANGING THE DRIVE LETTER

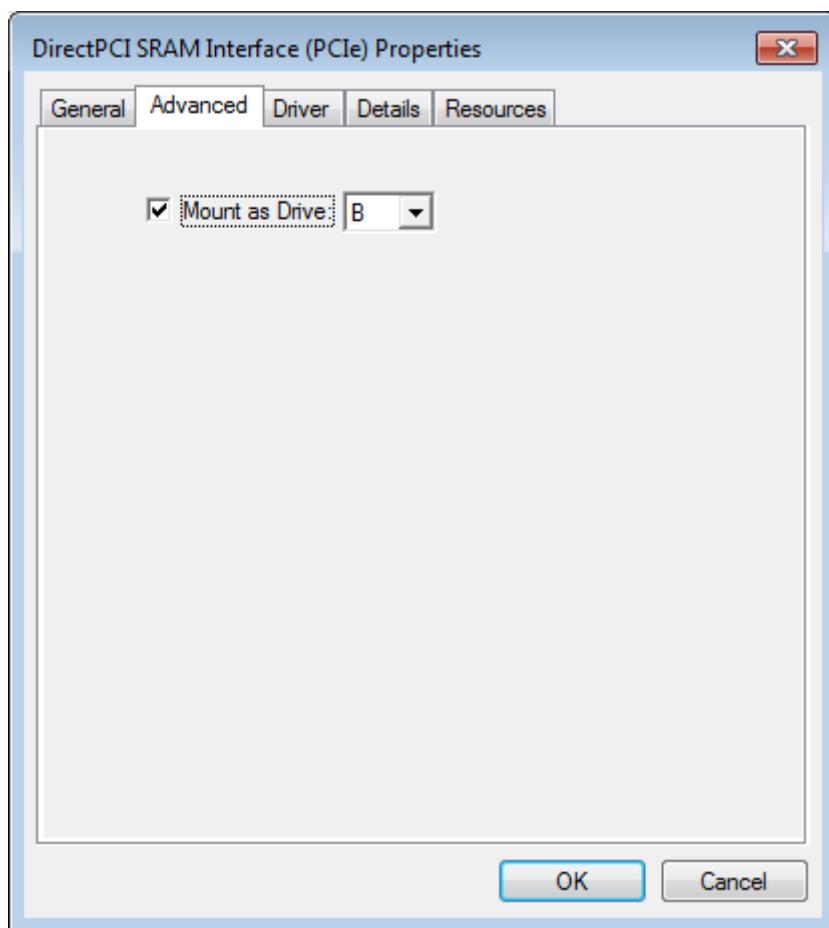
To change the drive letter, open the Device Manager application and locate the DirectPCI SRAM Interface or DirectPCI ROM Interface as desired.

Figure 19. Device Manager Window Showing DirectPCI SRAM and ROM Devices



Press the right mouse button to obtain the context menu and then select the Properties option. Now select the *Advanced* tab.

Figure 20. Advanced Properties Tab for the DirectPCI SRAM and ROM devices



To ensure that the device is given a drive letter, press the “Mount as Drive” button so that the tick shows; then select the drive letter required and press OK. To disable drive letter assignment, simply clear the “Mount as Drive” button.

#### **1.10.2 NOTE ABOUT DELAYS WHEN RESUMING/WAKING A SYSTEM WITH AN UNFORMATTED SRAM OR ROM DRIVE**

There may be some delay induced when resuming Windows from sleep states if a drive letter is assigned to the SRAM or ROM devices but the drives are not formatted.

The solution is to disable mounting either or both of the SRAM and ROM block devices if that device is not to be used as a drive.

Note that you might also observe this behavior if another unformatted medium is present in the system under similar circumstances.

## 1.11 NOTE ON SOFTWARE VERSION AND RELEASES

All software versions are identified by text in the format

*[v]A.B.C.D\_[special\_]stage\_OS-hardware*

Where each component is as follows:

Figure 21. Software Version Number Components

<b>Field</b>	<b>Significance</b>
A	This is the major version number. A change in this number indicates significant changes to the feature set, possibly incompatible with previous versions.
B	This is the minor version number. A change in this number indicates relatively minor changes to the feature set which are rarely incompatible with previous versions.
C	This is the scheduled maintenance update number. A change in this number indicates the presence in the release of fixes for known bugs. Scheduled maintenance releases typically do not include any new features.
D	This is the build number, which starts from 1 and increases monotonically as new builds of a given product are completed. Each build number is used once only per version number of a product across all supported hardware and OS combinations. Thus, for example, if build 3.0.0.72 is DirectPCI for 32-bit Microsoft Windows 7 Installations there can be no DirectPCI v3.0.0.72 for any other product.
Special	If present, this indicates that the given build is a <i>special</i> build of the product. The text <i>special</i> may indicate what this is; for example, a GLIBC v2.3 build has the value <i>glibc2.3</i> to distinguish it from other types.
Stage	<p>This indicates the stage of development of the product at the time the distribution was built:</p> <ul style="list-style-type: none"> <li>• <i>devel</i> – the product is undergoing development still and its state with regard to required features and fixes is indeterminate and not fully documented; most significant features may be missing and serious bugs may be present. The distribution has not had any formal QC testing.</li> <li>• <i>alpha</i> – this is the usual first level of release visible to customers. Most but not all significant features are present and some serious bugs are still present. Minimal QC time has been allocated to testing the release.</li> <li>• <i>beta</i> – this is an intermediate level which indicates all significant features are present and most serious bugs have been found and removed. Further QC time has been completed for this release.</li> <li>• <i>release</i> – this is the final level of quality determine and indicates that all defined features are present and no serious bugs have been found.</li> </ul>

<b>Field</b>	<b>Significance</b>
OS-hardware	<p>This indicates the target OS and hardware environment. Likely values are:</p> <ul style="list-style-type: none"> <li>• win32_i386</li> <li>• win7_i686</li> <li>• win7_x64</li> <li>• linux_i686</li> <li>• linux_x86_64</li> </ul>

## 1.12 NOTE ON DOCUMENTATION VERSIONS

Most new manuals for software products following a version numbering convention which more closely relates to the product version number to which the manually actually applies. The new format is:

*200-xxx-[v]A.B\_nn*

Where each component is as follows:

Figure 22. Documentation Version Number Components

<b>Field</b>	<b>Significance</b>
200-xxx	This is the document's individual code which identifies its subject area.
A	This is the major version number component for the version of the product to which the document applies.
B	This is the minor version number component for the version of the product to which the document applies. Only the two most significant components of the product version number are presented in the documentation version; this is because it is intended that ordinarily maintenance releases of software would not required significant documentation changes.
nn	This is a 1- or 2-digit number which identifies the revision of the manual for a given version.

Page Intentionally Blank

# 2

## THE DIRECTPCI APIs

The following sections document the facilities that DirectPCI provides and the programming activities for which you might need them.

### 2.1 WHICH API SUPPORTS WHICH GAMING FUNCTION?

#### 2.1.1 APIs GROUPED BY GAMING FUNCTION

The following table shows the available APIs by breaking them down according to typical gaming functions.

Figure 23. DirectPCI APIs Categorised by Gaming Function

Type of Activity	APIs or Functions	Notes
<b>Communications:</b> I2C, COM Ports, RS232, RS485 etc.	dpci_i2c Windows/Linux TTY APIs	For regular COM ports, please refer to the <i>uart_demo</i> program which is supplied in source code form with the SDK. This uses standard OS APIs available natively in both Linux and Windows OSs.
<b>Data Storage:</b> NVRAM, EEPROMs, metering and EGM counters, configuration	dpci_rom dpci_sram dpci_idprom dpci_e2	See 2.1.2. DirectPCI offers a range of facilities for storing and accessing data for a range of purposes from counter storage to license codes and configuration.
<b>Input / output:</b> digital inputs, outputs, lamps, buttons and switches.	dpci_io dpci_gpio	

Type of Activity	APIs or Functions	Notes
<b>Board Management:</b> batteries, power-failure detection, time and date, quiet mode, configuration	dpci_bat dpci_idlp dpci_pfd dpci_e130 dpci_qm dpci_ts (2.13 - p218) dpci_boards	
<b>Cabinet Security:</b> door-switches, lock boxes	dpci_idlp (2.8 - p123)	Door switches are typically attached to the INTRUS#0-INTRUS#7 inputs on typical DPX-series boards.
<b>Unique ID:</b> unique ID, serial number, licensing	dpci_idprom_readid (2.9.1 - p151) or dpci_core_get_serial (2.20.5 - p270)	The DirectPCI does not need any licensing but the unique IDs it makes available can be factored into platform authentication algorithms.
<b>Time-and-Date</b>	dpci_idlp (2.8 - p123)	
<b>Watchdogs</b>	dpci_wd (2.7 - p118)	
<b>Errors and Debugging</b>	Debug API Error API	
<b>Software and Firmware Versions</b>	SW Version Management API (2.5 - p97) dpci_id_fwversion (2.8.3 - p125)	
<b>GAT Requirements, verification</b>	dpci_bios_size (see 2.20.3) and dpci_bios_dump (see 2.20.4)	See also See 2.1.2.

## 2.1.2 STORAGE MEDIA ACCESSIBLE USING DIRECTPCI APIs

The following table illustrates the various methods supported in DirectPCI for storing data.

Figure 24. List of Devices Supporting Data Storage

Device	Size(s)	Notes

Device	Size(s)	Notes
<i>BIOS flash ROM</i>	1MB, 2MB, 4MB or 8MB	<p><b>Physical Format:</b> Removable PCB module, fitted as standard</p> <p><b>Fitment:</b> standard on all boards.</p> <p><b>Location:</b> Chipset memory controller hubs (FCH, PCH etc.) SPI BUS #0</p> <p><b>API:</b> dpci_bios (see 2.20.3 and 2.20.4)</p> <p><b>Usage:</b> The device contains the system BIOS and chipset firmware. The size and contents can be read so that they can be checksummed for the integrity checks required by many gaming jurisdictions.</p> <p><b>Notes:</b> The dpci_bios API allows a simple complete dump of the ROM to a memory buffer, which can then be written to a file.</p>
<i>DirectPCI ROM Extension</i>	1MB or 2MB	<p><b>Physical Format:</b> PLCC32 OTPROM, AT27M080</p> <p><b>Fitment:</b> optional</p> <p><b>Location:</b> DirectPCI FPGA Parallel bus</p> <p><b>API:</b> dpci_rom (2.3 - p81)</p> <p><b>Usage:</b> Platform verification code for execution under BIOS control prior to OS boot. Can be used for other read-only storage.</p> <p><b>Notes:</b> Suitable devices are becoming increasingly more expensive and harder to obtain, this storage option is now subject to planned obsolescence. Some earlier DPX-series motherboards support two individual 1MB chips. Please refer to the board's motherboard manual for further details.</p>
<i>DirectPCI I2C EEPROM</i>	32kB	<p><b>Physical Format:</b> Atmel AT24C256 soldered on board, Standard</p> <p><b>Fitment:</b> fitted on all DPX-S/C series and some DPX-E-series.</p> <p><b>Location:</b> DirectPCI FPGA I2C bus</p> <p><b>API:</b> dpci_eeprom (2.11 - p187)</p> <p><b>Usage:</b> This device has a write-protect jumper. This device is not fitted on all boards. Please refer to the board's motherboard manual for further details. This device has a relatively slow access speed.</p> <p><b>Notes:</b></p>
<i>DirectPCI SRAM</i>	1MB - 8MB	<p><b>Physical Format:</b> multiple SRAM chips soldered on board.</p> <p><b>Fitment:</b> standard on all boards. Sizes varies and is an option at ordering time.</p> <p><b>Location:</b> DirectPCI FPGA Parallel bus</p> <p><b>API:</b> dpci_sram (2.2 - p61)</p> <p><b>Usage:</b> This device is typically used to store a gaming machine's state data, including but not limited to: soft metering, last-game state, last game image and configuration. This device has fast write-access.</p> <p><b>Notes:</b> The SRAM memory can be mounted as a block device and formatted like a floppy disk.</p>

Device	Size(s)	Notes
<i>DirectPCI IDPROM</i>	128 bytes	<p><b>Physical Format:</b> Dallas Maxim DS2431 soldered on board</p> <p><b>Fitment:</b> All DPX-S/C series.</p> <p><b>Location:</b> DirectPCI FPGA OneWire SYSID BUS</p> <p><b>API:</b> dpci_idprom (2.9 - p151)</p> <p><b>Notes:</b></p> <p>This device is not fitted on all boards. Please refer to the board's motherboard manual for further details.</p> <p>This device has a write-lock which can be used to permanently protect the contents of the device.</p> <p>This device has a relatively slow access speed.</p>

### 2.1.3 HANDLING DIGITAL I/O AND DOOR SWITCHES

Many electronic gaming machines still use mechanical switches for buttons that game players require for regular play. In addition, most game cabinets are required to perform monitoring of access to the internal compartments of the machine's main cabinet in order to audit access to the main board, software installation and cash handling areas.

The DirectPCI APIs support access to the digital input lines (DI0-DI31) and the intrusion detection lines.

The following tables summarise access methods for handling digital input lines and intrusion detection circuits.

Figure 25. APIs for Handling Digital Inputs and Intrusion Circuits

API Name	Description
Event Streaming API 2.6 (p105)	<p>This API allows non-blocking access to a range of input sources, including both digital input lines (DI0-DI31) and intrusion monitoring events.</p> <p>One call can receive events for as many or as few input sources as required.</p> <p>The API supports event duplication via multiple streams in order to allow multiple event consumers to monitor reliably the same input devices.</p> <p>The API supports 'callback' functions which after registration by the customer code, will be called to receive new events, thus allowing fully asynchronous delivery.</p> <p>In combination with callbacks, de-bouncing of digital inputs is supported to filter noise from mechanical switches.</p>

API Name	Description
Digital I/O API 2.10 (p160)	<p>This is an older API dating from the earliest releases of the DirectPCI SDK.</p> <p>This <code>dpci_id_readevent()</code> function allows synchronous but non-blocking notification of changes on the digital input lines.</p> <p>Only one caller at a time across all processes and users can reliably use this API. Using this API may also conflict with users of the event streaming API.</p>
Intrusion Detection and Logging Processor API 2.8 (p123)	<p>This is an older API dating from the earliest releases of the DirectPCI SDK.</p> <p>This <code>dpci_id_readevent()</code> function allows synchronous but non-blocking notification of change on the intrusion input circuits) from the intrusion detection and logging processor (IDLP). The additional function <code>dpci_id_wait_event()</code> allows waiting a specified time for an IDLP event.</p> <p>Only one caller at a time across all processes and users can reliably use this API. Using this API may also conflict with users of the event streaming API.</p>

#### 2.1.4 COMPLETE LIST OF ALL DIRECTPCI APIs

The following table lists the DirectPCI APIs and their uses.

Figure 26. Description of All APIs

API Name	Section	Description	Demo Program
SRAM API	2.2 (p61)	This API allows access to the static (non-volatile), battery-backed RAM that exists on the DPX-series boards.	<code>sram_demo</code>
ROM API	2.3 (p81)	This API allows access to the contents of the optional ROM chip(s) that exists on many of the DPX-series boards.	<code>rom_demo</code>
Software Version Management API	2.5 (p97)	This API allows program access to determine the version(s) of DirectPCI software in use.	<code>io_demo</code>

API Name	Section	Description	Demo Program
Event Streaming API	2.6 (p105)	This API is new in v3.0 and is intended to replace the existing Digital I/O and IDLP API facilities that allow asynchronous access to events such as changes to an digital input or an event at the IDLP.	io_demo, callback_demo
Host Watchdog API	2.7 (p118)	This API provides control of the host watch-dog.	idlp_demo
Intrusion Detection and Logging Processor API	2.8 (p123)	This API provides complete access to the facilities provided by the IDLP processor.	idlp_demo
IDPROM API	2.9 (p151)	This API provides read access to the 128-byte IDPROM and its 64-bit unique ID.	io_demo
Digital I/O API	2.10 (p160)	This API provides complete access to the digital inputs and outputs.	io_demo
Motherboard EEPROM API	2.11 (p187)	This API provides access to the 32-kbyte EEPROM provided on most DPX-series motherboards.	eeprom_demo
I2C API	2.12 (p196)	This API provides a generic way to access I <sup>2</sup> C peripherals connected to the DirectPCI hardware's expander connectors (where available).	i2c_demo
Temperature Sensor API (I/O Board II)	2.13 (p218)	This API provides access to the temperature sensor on the 80-0062 I/O board for use on ConnectBus-II® systems.	ts_demo
Battery API	2.14 (p235)	This API provides access to the system batteries connected to the DirectPCI facility.	batt_demo
Error Handling API	2.15 (p247)	This API allows retrieval of error codes and error messages.	All demos
Quiet-mode API	2.16 (p250)	The API allows control of quiet-mode operation available on DPX-C and DPX-S series boards.	None

API Name	Section	Description	Demo Program
I/O Board API	2.17 (p253)	This API allows determination of the I/O board version/revision information and control of the I/O board watchdog on ConnectBus-II® systems.	io_demo
GPIO API	2.18 (p259)	The API allows control of GPIO pins available on DPX-series motherboards.	None
DPX-E130 Power Supply API	Backup 2.19 (p263)	This API is for the DPX-E130 board only and pertains to the back-up power (supercap) supply facility which holds the system alive to power-fail shutdown processing can take place.	None
Miscellaneous API	2.20 (p263)	This API collates a number of functions covering a range of facilities not documented elsewhere.	io_demo
Debug API	2.21 (p271)	This API allows programmatic control over debugging messages when the appropriate build of the library or driver is in use. See also Debugging (1.7 ).	
Board Identification API	2.22 (p280)	This API allows programmatic identification of the type of host system that the software is running on.	

The following shows the correlation between drivers and APIs and header files:

Figure 27. Drivers, Headers and Supported APIs

Driver	Header file(s)	APIs supported
dpci_mem	dpci_sram_api.h	SRAM API
	dpci_rom_api.h	ROM API
dpci_core	dpci_core_api.h	Software Version Management API
	dpci_core_hw.h	Event Streaming API
		Host Watchdog API
		Intrusion Detection and Logging Processor API
		IDPROM API
		Digital I/O API
		Motherboard EEPROM API
		I2C API
		Temperature Sensor API (I/O Board II)
		Battery API
		Error Handling API
		Quiet-mode API
		I/O Board API
		GPIO API
dpci_boards.h		DPX-E130 Backup Power Supply API
		Miscellaneous API
		Debug API
	dpci_boards.h	Board Identification API

## 2.2 SRAM API

Advantech Innocore boards currently support one or two banks (chips) of battery-backed 3.3V SRAM. The memory can be implemented using a range of chip densities supporting from 128KByte to 2048KByte chips, providing up to 8 Megabyte of storage. The memory is directly available on the PCI bus and thus can be directly mapped into the address space of a running process where host OS facilities support it.

The SRAM API is provided by the *dpci\_mem* driver and *libdpci* and supplies functions to allow the following:

- read or write access to any single byte (8-bit), word (16-bit) or dword (32-bit) location;
- read or write access to any arbitrarily chosen block of memory;
- the facility to directly map SRAM into process memory thereby allowing efficient access without the overhead of system calls;
- thread-safe access to all memory.

If the SRAM loses its supply voltage due to faulty battery then its contents are lost and the new contents of the SRAM are undefined.

The following outlines the files required for the SRAM API.

### 2.2.1 FILES FOR THE SRAM API

Figure 28. SRAM API and Driver Files

Header File	dpci_sram_api.h
Library Files (Linux)	libdpci.a (non-shared) libdpci.so (shared)
Library Files (Windows)	libdpci.dll (shared) libdpci.lib (shared) libdpci-static.lib (non-shared)
Driver Files (Linux)	dpci_mem.ko
Driver Files (Windows)	dpci_mem.sys dpci_mem.inf dpci_mem_amd64.cat (Windows 7) dpci_mem_x86.cat (Windows 7) ramdisk.dll
Device Files (Linux)	/dev/sram0 (for API use only) /dev/sramd0 (for SRAM disk)

Header File	dpci_sram_api.h
Device Files (Windows)	\.\B:

**Note:** all of the SRAM access API functions share a common file descriptor (Linux) or file handle (Windows) which is opened once and then never closed in order to remove as many OS latencies as possible. Users should avoid issuing code to close all open files in between SRAM API calls.

For SRAM API functions, thread-safety means that a given thread will complete an access to SRAM before another thread may start write-access to SRAM; however multiple threads may read SRAM simultaneously. (A multiple-reader, single-writer lock is used for efficiency.) Exceptions from this are noted for the memory-mapped access option.

**Note:**

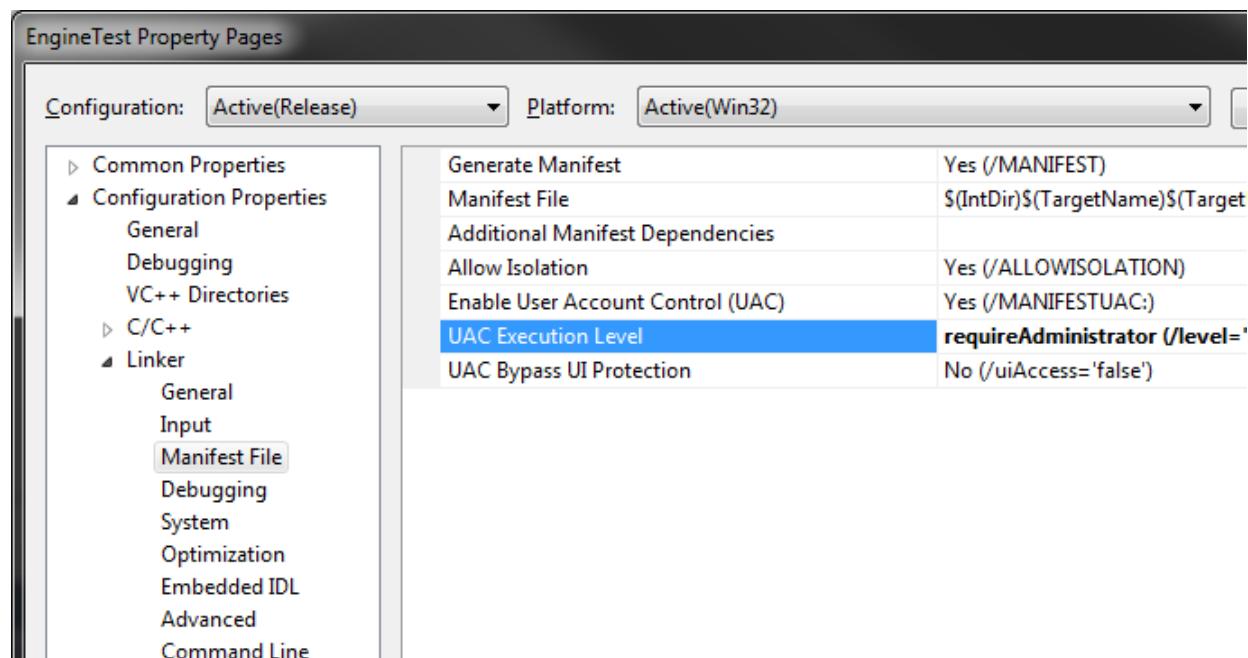
Under Windows 7, UAC controls mean that access to the underlying SRAM device must be executed with elevated (i.e. Administrator) privilege.

By default, the SRAM\_demo program binary is compiled so that Windows 7 will automatically prompt for privilege escalation via UAC.

Your own programs should do likewise by setting the UAC Execution Level setting in the manifest section of your Visual Studio project's settings thus:

The figure shows the location of this setting in Visual Studio 2010's properties dialog windows.

Figure 29. Setting the UAC Execution Level for Programs Accesing SRAM



You can also do this by including in the Manifest section of the Visual Studio project file the XML tag `<UACExecutionLevel>RequireAdministrator</UACExecutionLevel>`.

## 2.2.2 SRAM Disk Drive

### Definition (Windows)

```
C:>format B:
```

### Definition (Linux)

```
# mkfs.ext2 /dev/sramd0
# mount /dev/sramd0 /mnt
```

### Description

The SRAM Disk Drive facility allows the SRAM to be used as a virtual disk drive under Windows and Linux. Under Linux the driver can also be encrypted easily use the crypto-loop driver.

Under Windows, the SRAM automatically appears unformatted as drive B and so can be formatted with a FAT-style file system prior to files being written thereto.

Under Linux the dpci\_mem driver supports the device node `/dev/sramd0` which can be formatted and mounted on a directory in the file system.

When the SRAM is used to host a file system, it must not be accessed programmatically using the `dpci_sram_*()` functions.

The SRAM drive should be formatted once before first use; thereafter it need not be formatted again unless the contents become lost due to a programmatic access or loss of battery power.

### Notes

Under Linux, the `/dev/sram0` node cannot be used when creating or mounting a file system. Similarly, the `/dev/sramd0` node should not be used for direct access to the SRAM as accesses to it are buffered by the Linux operating system kernel.

Under Linux, the SRAM disk driver supports sub-partitioning. Partitions are typically given device nodes named `/dev/sramd0pN` where  $N$  is the partition number.

Under Windows, SRAM drive partitioning is not supported yet.

### Available

The SRAM *disk-drive* feature was first available in DirectPCI SDK & Run-time v2.2.0.

### Example

Under Linux, it is possible to encrypt the SRAM drive thus:

```
#!/bin/bash
# Load the device drivers we will be relying on
#
modprobe dm_mod
modprobe dm_crypt
modprobe aes
```

```
# establish a crypto mapping through a device-mapper block device.  
#  
blksize=$(blockdev --getsize /dev/sramd0)  
key="0123456789abcdef0123456789abcdef" # 32 hex digits = 128-bit  
echo "0 $blksize crypt aes-plain $key 0 /dev/sramd0 0" | dmsetup create  
crypto_mapped_device  
  
mkfs.vfat /dev/mapper/crypto_mapped_device  
mount /dev/mapper/crypto_mapped_device /mnt/sram_disk
```

## 2.2.3 GET SIZE OF SRAM

### Definition

```
#include <dpci_sram_api.h>

int dpci_sram_size(void);
```

### Parameters

None

### Description

This function returns the size of the SRAM in bytes.

If no SRAM is configured then ordinarily the device drivers for the SRAM function will not be loaded by the operating system. This in turn means the device nodes files for the device driver do not exist, causing attempts to open them to fail with a “file not found” error: under Linux *errno* is set to ENOENT, under Microsoft Windows XP the function GetLastError() returns ERROR\_FILE\_NOT\_FOUND

### Return Value

- |     |   |
|-----|---|
| >=0 | The size of the SRAM in bytes.              |
| -1  | No SRAM is configured or an error occurred. |

### Available

The *dpci\_sram\_size()* function was added in DirectPCI SDK & Run-time v2.0.0.

### Example

```
#include <string.h>
#include <errno.h>
#include <dpci_sram_api.h>
...
int size = dpci_sram_size();
if (size == -1)
{
    if (errno == ENOENT)
    {
        size = 0;
    }
    else
    {
        fprintf(stderr,
                "SRAM size could not be determined: %s\n",
                strerror(errno));
    }
}
```

A further example can be found in the *sram\_demo.c* program.

## 2.2.4 READ BYTE FROM SRAM

### Definition

```
#include <dpci_sram_api.h>

int dpci_sram_read8(unsigned int offset, unsigned char *pvalue);
```

### Parameters

offset	The offset from the beginning of the first bank of SRAM.
pvalue	The memory location into which to copy the requested data.

### Description

Reads an 8-bit item of data from the SRAM interface at location *offset* and places it into user memory at *pvalue*. It is an error to specify a SRAM offset for a location not supported in physical hardware.

### Return Value

0	Success
-1	An error occurred

### Example

```
#include <stdio.h>
#include <dpci_sram_api.h>

...
unsigned char data;
if (dpci_sram_read8(0, &data) == -1)
{
    printf("Unable to read byte from SRAM.\n");
}
```

A further example can be found in the *sram\_demo.c* program.

## 2.2.5 READ WORD FROM SRAM

### Definition

```
#include <dpci_sram_api.h>

int dpci_sram_read16(unsigned int offset, unsigned short *pvalue);
```

### Parameters

offset	The offset in memory to read from.
pvalue	The memory location into which to copy the requested data.

### Description

Reads a 16-bit item of data from the SRAM interface at location *offset* and places it into user memory at *pvalue*. It is an error to specify a SRAM offset for a location not supported in physical hardware.

---

**Please note – The DirectPCI hardware does not support 16-bit accesses not aligned to a 16-bit word boundary. Accesses to such locations are treated as though the least significant bit (bit 0) were zero.**

---

### Return Value

0	Success
-1	An error occurred

### Example

```
#include <stdio.h>
#include <dpci_sram_api.h>
...
unsigned short data;
if (dpci_sram_read16(0, &data) == -1)
{
    printf("unable to read word from SRAM.\n");
}
```

A further example can be found in the *sram\_demo.c* program.

## 2.2.6 READ DWORD FROM SRAM

### Definition

```
#include <dpci_sram_api.h>

int dpci_sram_read32(unsigned int offset, unsigned int *pvalue);
```

### Parameters

offset	The offset in memory to read from.
pvalue	The memory location into which to copy the requested data.

### Description

Reads a 32-bit item of data from the SRAM interface at location *offset* and places it into user memory at *pvalue*. It is an error to specify a SRAM offset for a location not supported in physical hardware.

#### Note for DPX-112, DPX-116 and DPX-117 users

**While the functionality is provided in the API, Innocore recommends that 32-bit accesses should not be made to the SRAM, as the timing does not strictly meet the requirements of the PCI Specification. Please see the relevant motherboard hardware manual for more details.**

**Please note – The DirectPCI hardware does not support 32-bit accesses not aligned to a 32-bit word boundary. Accesses to such locations are treated as though the two least significant bits (bits 0 and 1) were zero.**

### Return Value

0	Success
-1	An error occurred

### Example

```
#include <stdio.h>
#include <dpci_sram_api.h>
...
unsigned int data;
if (dpci_sram_read32(0, &data) == -1)
{
    printf("Unable to read dword from SRAM.\n");
}
```

A further example can be found in the *sram\_demo.c* program.

## 2.2.7 READ BLOCK FROM SRAM

### Definition

```
#include <dpci_sram_api.h>

int dpci_sram_read(unsigned int offset,
                    void *buffer,
                    unsigned int datalen)
```

### Parameters

offset	The offset in memory to read from.
buffer	Pointer to memory address to store the data returned by the function.
datalen	The length of the data to be read from SRAM.

### Description

Reads *datalen* consecutive bytes of data from the DirectPCI SRAM interface at the offset *offset* and copies into user memory defined by *buffer*.

The semantics of addressing are consistent with Unix/POSIX read/write system calls: if the inclusive memory range [offset:offset+datalen-1] should include byte locations beyond the range physically supported, then only as many bytes are transferred as actually are possible and the transfer is truncated, the return code indicating the number of bytes actually transferred. An error code is not returned.

For example, an attempt to read 8 bytes of data from SRAM starting 4 bytes from the highest available memory address will transfer only four bytes and return the value 4.

### Return Value

>=0	Number of bytes read
-1	An error occurred

### Example

```
#include <stdio.h>
#include <dpci_sram_api.h>

...
char buffer [33];
int bytes_got;

bytes_got = dpci_sram_read(0, buffer, sizeof(buffer) - 1);
if (bytes_got < sizeof(buffer))
{
    printf("Couldn't read whole buffer from SRAM.\n");
}
else
```

```
{  
    buffer[sizeof(buffer) - 1] = '\0';  
    printf("Buffer is %s\n", buffer);  
}
```

A further example can be found in the sram\_demo.c program.

## 2.2.8 WRITE BYTE TO SRAM

### Definition

```
#include <dpci_sram_api.h>

int dpci_sram_write8(unsigned int offset, unsigned char value);
```

### Parameters

offset	The offset in memory to write to.
value	The value to write to the SRAM.

### Description

This function writes the 8-bit quantity *value* to the specified SRAM interface location *offset*. It is an error to specify a SRAM offset for a location not supported in physical hardware.

### Return Value

0	Operation completed successfully.
-1	Failure writing to SRAM.

### Example

```
#include <stdio.h>
#include <dpci_sram_api.h>

...

char data = 0x61;
if (dpci_sram_write8(0,data) == -1)
{
    printf("Unable to write byte to SRAM.\n");
}
else
{
    printf("SRAM successfully updated\n");
}
```

A further example can be found in the *sram\_demo.c* program.

## 2.2.9 WRITE WORD TO SRAM

### Definition

```
#include <dpci_sram_api.h>

int dpci_sram_write16(unsigned int offset, unsigned short value);
```

### Parameters

offset	The offset in memory to write to.
value	The value to write to the SRAM.

### Description

This function writes the 16-bit quantity *value* to the specified SRAM location *offset*. It is an error to specify a SRAM offset for a location not supported in physical hardware.

---

**Please note – The DirectPCI hardware does not support 16-bit accesses not aligned to a 16-bit word boundary. Accesses to such locations are treated as though the least significant bit (bit 0) were zero.**

---

### Return Value

0	Operation completed successfully.
-1	Failure writing to SRAM.

### Example

```
#include <stdio.h>
#include <dpci_sram_api.h>
...
unsigned short data = 0x61;
if (dpci_sram_write16(0, data) == -1)
{
    printf("Unable to write word to SRAM.\n");
}
else
{
    printf("SRAM successfully updated\n");
}
```

A further example can be found in the *sram\_demo.c* program.

## 2.2.10 WRITE DOUBLE WORD TO SRAM

### Definition

```
#include <dpci_sram_api.h>

int dpci_sram_write32(unsigned int offset, unsigned int value);
```

### Parameters

offset	The offset in memory to write to.
value	The value to write to the SRAM.

### Description

Writes the 32-bit quantity *value* to the specified SRAM location *offset*. It is an error to specify a SRAM offset for a location not supported in physical hardware.

#### Note for DPX-112, DPX-116 and DPX-117 users

**Please note – Although functionality is provided in the API to perform 32-bit accesses, Innocore recommends such accesses not be made to the SRAM, as the timing does not strictly meet the requirements of the PCI Specification. Please see the DirectPCI hardware manual for more details.**

**Please note – The DirectPCI hardware does not support 32-bit accesses not aligned to a 32-bit word boundary. Accesses to such locations are treated as though the two least significant bits (bits 0 and 1) were zero.**

### Return Value

0	Operation completed successfully.
-1	Failure writing to SRAM.

### Example

```
#include <stdio.h>
#include <dpci_sram_api.h>
...
unsigned int data = 0x61;
if (dpci_sram_write32(0,data) == -1)
{
    printf("Unable to write double word to SRAM.\n");
    exit(0);
}
printf("SRAM successfully updated\n");
```

A further example can be found in the *sram\_demo.c* program.

## 2.2.11 WRITE BLOCK TO SRAM

### Definition

```
#include <dpci_sram_api.h>

int dpci_sram_write(unsigned int offset,
                     void *buffer,
                     unsigned int datalen)
```

### Parameters

offset	The start offset in memory to write to.
*buffer	Pointer to memory address containing data to write to SRAM.
datalen	The length of the data to be written to SRAM.

### Description

Writes *datalen* consecutive bytes of data from the DirectPCI SRAM interface at the offset *offset* and copies into user memory defined by *buffer*.

The semantics of addressing are consistent with Unix/Posix read/write system calls: if the inclusive memory range [offset:offset+datalen-1] should include byte locations beyond the range physically supported, then only as many bytes are transferred as actually are possible and the transfer is truncated, the return code indicating the number of bytes actually transferred. An error code is not returned.

For example, an attempt to write 8 bytes of data to SRAM starting 4 bytes from the highest available memory address will transfer only four bytes and return the value 4.

### Return Value

>=0	The number of bytes written successfully to SRAM.
-1	An error occurred.

### Example

```
#include <stdio.h>
#include <dpci_sram_api.h>

char buffer [10] = {0,1,2,3,4,5,6,7,8,9};

if (dpci_sram_write(0, buffer, sizeof(buffer)) !=
    sizeof(buffer))
{
    printf("unable to write buffer to SRAM.\n");
}
```

A further example can be found in the *sram\_demo.c* program.

## 2.2.12 MEMORY MAP SRAM

### Definition

```
#include <dpci_sram_api.h>

void *dpci_sram_map()
```

### Parameters

None

### Description

Provides a pointer to a map of the SRAM in memory. The SRAM memory may be accessed directly using this pointer from user-space.

### Return Value

NULL	An error occurred
Non-NULL	Starting address of the memory map of SRAM

### Note

Using `memcpy()`, `strcpy()` and similar functions to read or write data from/to the SRAM map may not work reliably and should be avoided, as many standard C versions of these functions expect support for unaligned memory access. Care must be taken to use alternative implementations of `memcpy()` etc where needed.

The memory map SRAM API is provided only as an additional option to the user. It is advisable wherever possible to use the read/write APIs for SRAM access; as no thread-safety is implicit when accessing mapped SRAM directly. The application developer must implement explicit memory locking when using this facility.

### Availability

The `dpci_sram_map` function was added in v2.2.0.

The `dpci_sram` memory mapping is supported on these boards: DPX-S435, DPX-S430, DPX-S425, DPX-S420, DPX-S410, DPX-S415, DPX-E130, DPX-E120, DPX-SC710.

## Example

```
#include <stdio.h>
#include <dpci_sram_api.h>
...
char buffer [10] = {0,1,2,3,4,5,6,7,8,9};
char *map_ptr = NULL;
char *test_ptr = NULL;
int cnt = 10;

map_ptr = dpci_sram_map();
if (map_ptr == NULL)
{
    printf("Unable to map SRAM.\n");
}
else
{
    test_ptr = map_ptr;
    // write a string to the SRAM and read it back from the map
    if (dpci_sram_write(0, buffer, sizeof(buffer)) !=
        sizeof(buffer))
        printf("Unable to write buffer to SRAM.\n");
    else
    {
        while(cnt--)
            printf("%c", test_ptr++);
    }
}
```

## 2.2.13 UNMAP SRAM

### Definition

```
#include <dpci_sram_api.h>

int dpci_sram_unmap()
```

### Parameters

None

### Description

Unmaps SRAM memory thus invalidating the pointer previously returned on a map request.

### Return Value

-1	An error occurred
0	SRAM unmapped successfully

### Availability

The dpci\_sram\_unmap function was added in v2.2.0.

### Example

```
#include <stdio.h>
#include <dpci_sram_api.h>

...
if (dpci_sram_unmap() == -1)
{
    printf("Unable to unmap SRAM.\n");
}
```

## 2.2.14 MEMORY MAPPED SRAM (LINUX ONLY)

### Linux Definition

```
#include <sys/mman.h>

int fd;
void *addr;

fd = open("/dev/sram0", O_RDWR);
addr = mmap(NULL,
            dpci_sram_size(),
            PROT_READ | PROT_WRITE,
            MAP_SHARED,
            fd,
            0);
```

### Description

Support for the Linux *mmap()* system call is available to allow user to directly map SRAM memory into user process space. This is one of the most efficient ways to access SRAM contents as the latencies incurred by kernel calls are removed. The developer should read the Linux manual page for the *mmap()* call for details on how *mmap()* works before using this option.

Once the SRAM is mapped into user space, the memory can be accessed like any other memory using C language pointers.

### Notes

It is possible to have read-access only to the SRAM contents if the protection mode passed to *mmap()* is only PROT\_READ. The *mmap()* call will fail if PROT\_WRITE is used when the file descriptor was opened using O\_RDONLY.

The MAP\_PRIVATE flag is also supported – this can be used to take a snap-shot of the SRAM contents – changes made by the program to the mapped copy of the SRAM are not written back to the SRAM itself.

It is now preferable to use the *dpci\_sram\_map()* function instead of calling *mmap* directly. However, if you specifically need to map only a small amount of SRAM then *mmap()* may be more suitable.

There is no requirement to map the whole SRAM contents as shown in the example.

### See Also

[Linux \*mmap\(\)\* manual page \(manual section 2\)](#)

## Example

```
#include <stdio.h>
#include <sys/mman.h>
#include <dpci_sram_api.h>

...
int fd;
void *addr;
struct my_header
{
    unsigned int magic;
    time_t time;
    int version_code;
} *my_header_p;

/*
 * Open the device first.
 */
fd = open("/dev/sram0", O_RDWR);
if (fd == -1)
{
    fprintf(stderr,
            "Cannot open SRAM device: %s\n",
            strerror(errno));
    exit(1);
}

/*
 * Now map the whole SRAM contents.
 */
addr = mmap(NULL,
            dpci_sram_size(),
            PROT_READ | PROT_WRITE,
            MAP_SHARED,
            fd,
            0);
if (addr == (void *)-1)
{
    fprintf(stderr,
            "Cannot map SRAM device: %s\n",
            strerror(errno));
    exit(1);
}
printf("SRAM mapped to address 0x%08x\n", addr);

/*
 * Now print the SRAM header.
 */
my_header_p = (struct my_header *)addr;
printf("Header contents: \n");
printf("Magic number: %x\n", my_header_p->magic);
printf("Time stamp: %x\n", my_header_p->time);
printf("Version code: %x\n", my_header_p->version_code);
...
```

## 2.3 ROM API

Some applications may require access to unchangeable data from a non-volatile memory device. The DPX-series boards cater for this need by the provision of one or two 32-pin PLCC sockets for the installation of either one or two 8MBit (1MByte) EPROM/OTPROM chips. Please see the appropriate motherboard hardware manual for further details.

The ROM memory is directly available on the PCI bus and thus can be directly mapped into the address space of a running process where host OS facilities support it.

The ROM API is provided by the *dpci\_rom* driver and *libdpci* library; together these supply functions to allow the following:

- read access to any single byte (8-bit), word (16-bit) or dword (32-bit) location;
- read access to any arbitrarily chosen block of memory;
- a facility to directly map ROM into process memory thereby allowing efficient access without the overhead of system calls.

### 2.3.1 FILES FOR THE ROM API

The following outlines the files required for the ROM API.

Figure 30. ROM API and Driver Files

Header File	<i>dpci_rom_api.h</i>
Library Files (Linux)	<i>libdpci.a</i> (non-shared) <i>libdpci.so</i> (shared)
Library Files (Windows)	<i>libdpci.dll</i> (shared) <i>libdpci-static.lib</i> (shared) <i>libdpci.lib</i> (non-shared)
Driver Files (Linux)	<i>dpci_mem.ko</i>
Driver Files (Windows)	<i>dpci_rom.sys</i> <i>dpci_ rom.inf</i> <i>dpci_ rom_x86.cat</i> (Windows 7) <i>dpci_ rom_amd64.cat</i> (Windows 7) <i>ramdisk.dll</i>
Device Files (Linux)	<i>/dev/rom0</i> (for API use only) <i>/dev/romd0</i> (for SRAM disk)
Device Files (Windows)	<i>\.\R:</i>

---

**Note that all of the ROM access API functions share a common file descriptor (Linux) or file handle (Windows) which is opened once and then never closed in order to remove as many OS latencies as possible. Users should avoid issuing code to close all open files in between ROM API calls.**

---

## 2.3.2 GET SIZE OF ROM

### Definition

```
#include <dpci_rom_api.h>

int dpci_rom_size(void);
```

### Parameters

none

### Description

This function returns the size of the ROM in bytes. Where the ROM function is enabled this will always be 2MByte (2097152 bytes) or 64MByte (only on the legacy DPX-114 and DPX-115 products) because this is the maximum size supported by the ROM interface hardware. Note that it is not possible to determine when a smaller ROM chip has been fitted – in such cases memory-aliasing will occur.

If no ROM is configured in hardware then ordinarily the device drivers for the ROM function will not be loaded by the operating system. This in turn means the device nodes files for the device driver do not exist, causing attempts to open them to fail with a “file not found” error: under Linux *errno* is set to ENOENT, under Microsoft Windows XP the function GetLastErrorHandler() returns ERROR\_FILE\_NOT\_FOUND)

### Return Value

The size of the ROM in bytes.

If no ROM is configured or an error occurred, then -1 is returned.

### Notes

The `dpci_rom_size()` function first appeared in the DirectPCI API version 2.0.0.

## Example

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <dpci_rom_api.h>

...
int size;
size = dpci_rom_size();
if (size == -1)
{
    if (errno == ENOENT)
    {
        size = 0;
    }
    else
    {
        fprintf(stderr,
                "ROM size could not be determined: %s\n",
                strerror(errno));
    }
}
```

A further example can be found in the *rom\_demo.c* program.

## 2.3.3 READ BYTE FROM ROM

### Definition

```
#include <dpci_rom_api.h>

int dpci_rom_read8(unsigned int offset, unsigned char *pvalue);
```

### Parameters

offset	The offset from the beginning of the first bank of ROM.
pvalue	The memory location into which to copy the requested data.

### Description

Reads an 8-bit item of data from the ROM interface at location *offset* and places into user memory at *pvalue*. It is an error to specify a ROM offset for a location not supported in physical hardware.

### Return Value

0	Success
-1	An error occurred

### Example

```
#include <stdio.h>
#include <dpci_rom_api.h>

...
unsigned char data;
if (dpci_rom_read8(0, &data) == -1)
{
    printf("Unable to read byte from ROM.\n");
```

A further example can be found in the *rom\_demo.c* program.

## 2.3.4 READ WORD FROM ROM

### Definition

```
#include <dpci_rom_api.h>

int dpci_rom_read16(unsigned int offset, unsigned short *pvalue);
```

### Parameters

offset	The offset in memory to read from.
pvalue	The memory location into which to copy the requested data.

### Description

Reads a 16-bit item of data from the ROM interface at location *offset* and places into user memory at *pvalue*. It is an error to specify a ROM offset for a location not supported in physical hardware.

---

#### Note for DPX-112, DPX-116 and DPX-117 users

**The DirectPCI hardware does not support 16-bit accesses not aligned to a 16-bit word boundary. Accesses to such locations are treated as though the least significant bit (bit 0) were zero.**

---

### Return Value

0	Success
-1	An error occurred

### Example

```
#include <stdio.h>
#include <dpci_rom_api.h>

...
unsigned short data;
if (dpci_rom_read16(0, &data) == -1)
{
    printf("unable to read word from ROM.\n");
}
```

A further example can be found in the *rom\_demo.c* program.

## 2.3.5 READ DWORD FROM ROM

### Definition

```
#include <dpci_rom_api.h>

int dpci_rom_read32(unsigned int offset, unsigned int *pvalue);
```

### Parameters

offset	The offset in memory to read from.
pvalue	The memory location into which to copy the requested data.

### Description

Reads a 32-bit item of data from the ROM interface at location *offset* and places into user memory at *pvalue*. It is an error to specify a ROM offset for a location not supported in physical hardware.

#### Note for DPX-112, DPX-116 and DPX-117 users

While the functionality is provided in the API, Innocore recommends that 32-bit accesses should not be made to the ROM, as the timing does not strictly meet the requirements of the PCI Specification. Please see the relevant motherboard hardware manual for more details.

Please note – The DirectPCI hardware does not support 32-bit accesses not aligned to a 32-bit word boundary. Accesses to such locations are treated as though the two least significant bits (bits 0 and 1) were zero.

### Return Value

0	Success
-1	An error occurred

### Example

```
#include <stdio.h>
#include <dpci_rom_api.h>
...
unsigned int data;
if (dpci_rom_read32(0, &data) == -1)
{
    printf("unable to read dword from ROM.\n");
}
```

A further example can be found in the *rom\_demo.c* program.

## 2.3.6 READ BLOCK FROM ROM

### Definition

```
#include <dpci_rom_api.h>

int dpci_rom_read( unsigned int offset,
                   void *buffer,
                   unsigned int datalen)
```

### Parameters

offset	The offset in memory to read from.
buffer	Pointer to memory address to store the data returned by the function.
datalen	The length of the data (in bytes) to be read from ROM.

### Description

Reads *datalen* consecutive bytes of data from the DirectPCI ROM facility at the ROM offset *offset* and copies into user memory defined by *buffer*.

The semantics of addressing are consistent with Unix/Posix read/write system calls: if the inclusive memory range [offset:offset+datalen-1] should include byte locations beyond the range physically supported, then only as many bytes are transferred as actually are possible and the transfer is truncated, the return code indicating the number of bytes actually transferred. An error code is not returned.

For example, an attempt to read 8 bytes of data from ROM starting 4 bytes before the highest available memory address will transfer only four bytes and return the value 4.

### Return Value

>=0	Number of bytes read
-1	An error occurred

### Example

```
#include <stdio.h>
#include <dpci_rom_api.h>

char buffer [33];

int bytes_got = dpci_rom_read(0, buffer, sizeof(buffer) - 1);
if (bytes_got < sizeof(buffer))
{
    printf("Couldn't read whole buffer from ROM.\n");
    exit(1);
}
buffer[sizeof(buffer) - 1] = '\0';
```

```
printf("Buffer is %s\n", buffer);
```

A further example can be found in the *rom\_demo.c* program.

## 2.3.7 MEMORY MAP ROM

### Definition

```
#include <dpci_rom_api.h>

void *dpci_rom_map()
```

### Parameters

None

### Description

Provides a pointer to a map of the ROM in memory. The ROM memory may be accessed directly using this pointer from user-space.

### Return Value

NULL	An error occurred
Non-NULL	Starting address of the memory map of ROM

### Note

Using `memcpy()` to read data from the ROM map may not work reliably as not all versions of `memcpy()` support unaligned memory access. Hence, care should be taken to use alternative implementations of `memcpy` where applicable.

This API is provided only as an additional option to the user. It is advisable to use the read APIs for ROM access wherever possible.

### Availability

The `dpci_rom_map` function was added in v2.2.0.

The `dpci_rom` memory mapping is supported on these boards: DPX-S435, DPX-S430, DPX-S425, DPX-S420, DPX-S410, DPX-S415, DPX-E130, DPX-E120, DPX-SC710.

## Example

```
#include <stdio.h>
#include <dpci_rom_api.h>
...
char buffer [10] = {0,1,2,3,4,5,6,7,8,9};
char *map_ptr = NULL;
char *test_ptr = NULL;
int cnt = 10;

map_ptr = dpci_rom_map();
if (map_ptr == NULL)
{
    printf("Unable to map ROM.\n");
}
else
{
    test_ptr = map_ptr;

    /* read a string from the ROM using read API and cross-check with
read_from_ROM map */
    if (dpci_rom_read(0, buffer, sizeof(buffer)) != sizeof(buffer))
        printf("Unable to read string from ROM.\n");
    else
    {
        while(cnt--)
        {
            if (*buffer++ != *test_ptr++)
                printf("ROM Map data differs from read data\n");
        }
    }
}
```

## 2.3.8 UNMAP ROM

### Definition

```
#include <dpci_rom_api.h>

int dpci_rom_unmap()
```

### Parameters

None

### Description

Unmaps ROM memory thus invalidating the pointer previously returned on a map request.

### Return Value

-1	An error occurred
0	ROM unmapped successfully

### Availability

The `dpci_rom_unmap()` function first appeared in the DirectPCI API version 2.2.0.

### Example

```
#include <stdio.h>
#include <dpci_rom_api.h>

...
if (dpci_rom_unmap() == -1)
{
    printf("Unable to unmap ROM.\n");
```

## 2.3.9 MEMORY MAPPED ROM (LINUX ONLY)

### Linux Definition

```
#include <sys/mman.h>

int fd;
void *addr;

fd = open("/dev/rom0", O_RDWR);
addr = mmap(NULL,
            rom_size(),
            PROT_READ,
            MAP_SHARED,
            fd,
            0);
```

### Description

Support for the Linux *mmap()* system call is available to allow user to directly map ROM memory into user process space. This is the most efficient way to access ROM contents as there are very few latencies incurred by kernel routines. The developer should read the Linux manual page for the *mmap()* call for details on how *mmap()* works before using this option.

Once the ROM is mapped into user space, the memory can be accessed like any other memory using C language pointers.

### Notes

There is no difference between a private or shared mapping for the ROM device.

It is now recommended to use the *dpci\_rom\_map()* function instead of calling *mmap* directly.

There is no requirement to map the whole ROM contents as shown in the example.

### See Also

[Linux \*mmap\(\)\* manual page \(manual section 2\)](#)

## Example

```
#include <stdio.h>
#include <sys/mman.h>
#include <dpci_rom_api.h>

...
int fd;
void *addr;
struct my_header
{
    unsigned int magic;
    time_t time;
    int version_code;
} *my_header_p;

/*
 * Open the device first.
 */
fd = open("/dev/rom0", O_RDONLY);
if (fd == -1)
{
    fprintf(stderr,
            "Cannot open SRAM device: %s\n",
            strerror(errno));
    exit(1);
}

/*
 * Now map the whole ROM contents.
 */
addr = mmap(NULL,
            dpci_rom_size(),
            PROT_READ,
            MAP_SHARED,
            fd,
            0);
if (addr == (void *)-1)
{
    fprintf(stderr,
            "Cannot map ROM device: %s\n",
            strerror(errno));
    exit(1);
}
printf("ROM mapped to address 0x%08x\n", addr);

/*
 * Now print the ROM header.
 */
my_header_p = (struct my_header *)addr;
printf("Header contents: \n");
printf("Magic number: %x\n", my_header_p->magic);
printf("Time stamp: %x\n", my_header_p->time);
printf("Version code: %x\n", my_header_p->version_code);
...
```

## 2.4 DIRECTPCI CORE I/O DRIVER

The DirectPCI *dpci\_core* driver provides services for *libdpci*, allowing access to all functions pertaining to the following:

- Digital outputs, digital inputs and detection of changes on digital inputs (Digital I/O API)
- Host Watchdog
- The Intrusion Detection and Logging Processor (IDLP) and its many functions
- The Event Streaming
- The on-board EEPROM
- I<sup>2</sup>C buses
- Temperature sensors on the I/O Board II (80-0062) I/O board.
- Version control,
- I/O Board
- IDPROM
- Batteries
- Quiet-mode
- Error handling
- GPIO

The following outlines the files required for the DPCI API.

### 2.4.1 FILES FOR THE CORE API

Figure 31. DPCI API and Core Driver Files

Header File	dpci_core_api.h
Library Files (Linux)	libdpci.a (non-shared) libdpci.so (shared)
Library Files (Windows)	libdpci.dll (shared) libdpci-static.lib (shared) libdpci.lib (non-shared)
Driver Files (Linux)	dpci_core.ko

Header File	dpci_core_api.h
Driver Files (Windows)	dpci_core.sys dpci_core.inf dpci_core_x86.cat (Windows 7) dpci_core_amd64.cat (Windows 7) dpci_multi.inf (for systems with on-board I/O) dpci_multi_x86.cat (for systems with on-board I/O) dpci_multi_amd64.cat (for systems with on-board I/O) dpci_80-1003.inf (for systems with 80-1003 I/O board) dpci_80-0062.inf (for systems with 80-0062 I/O board) dpci_80-1003_x86.cat (for systems with 80-1003 I/O board) dpci_80-0062_x86.cat (for systems with 80-0062 I/O board) dpci_80-1003_amd64.cat (for systems with 80-1003 I/O board) dpci_80-0062_amd64.cat (for systems with 80-0062 I/O board)
Device Files (Linux)	/dev/dpci0 (for API use only)
Device Files (Windows)	\.\DPCI

The most common DirectPCI hardware functions are implemented in user-space for easy access by the DirectPCI Software API.

---

**Note: all of the I/O access API functions share a common file descriptor (Linux) or file handle (Windows) which is opened once and then never closed in order to remove as many OS latencies as possible. Users should avoid issuing code to close all open files in between I/O API calls.**

---

## 2.5 SOFTWARE VERSION MANAGEMENT API

The software version management API is a general API to allow access to the versions of drivers and libraries that are in use on the host system.

### 2.5.1 GET API VERSION CODE

#### Definition

```
#include <dpci_core_api.h>

#define DPCI_VERSION_MAJOR(ver) (((ver) & 0xff000000) >> 24)
#define DPCI_VERSION_MINOR(ver) (((ver) & 0x00ff0000) >> 16)
#define DPCI_VERSION_MICRO(ver) (((ver) & 0x0000ff00) >> 8)
#define DPCI_VERSION_BUILD(ver) ((ver) & 0x000000ff)
int dpci_api_version(void);
```

#### Parameters

None.

#### Description

This function returns the version code of the API library currently in use by the caller. The version code is made of three components: the major number (bits 24-31), the minor number (bits 16-23) and the micro number

This function is provided mainly to aid debugging and distribution management.

#### Return Value

- |    |   |
|----|---|
| >0 | The operation completed successfully.                               |
| -1 | A problem occurred whilst trying to obtain the version information. |

#### Notes

The version code does not include information relating to the type of release, e.g. alpha, beta, final. See *dpci\_api\_version\_string()* for this information.

There are no separate versioning functions for the SRAM and ROM functions since the API and driver for these facilities will be almost always the same version as that for the DirectPCI I/O function.

The *dpci\_api\_version()* function first appeared in the DirectPCI API version 1.2.1.

#### See Also

- [dpci\\_api\\_version\\_string\(\) p.99](#)
- [dpci\\_drv\\_version\(\) p.100](#)

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
int version_code = dpci_api_version();
if (version_code != -1)
{
    printf("API Version is %d.%d.%d build %d\n",
        DPCI_VERSION_MAJOR(version_code),
        DPCI_VERSION_MINOR(version_code),
        DPCI_VERSION_MICRO(version_code),
        DPCI_VERSION_BUILD(version_code));
}
```

A further example can be found in the *io\_demo.c* program.

## 2.5.2 GET API VERSION STRING

### Definition

```
#include <dpci_core_api.h>

const char *dpci_api_version_string(void);
```

### Parameters

None.

### Description

This function returns the textual version string of the API library currently in use by the caller. The string returned includes a representation of the type of release, e.g. alpha, beta etc.

This function is provided mainly to aid debugging and distribution management.

### Return Value

>0	The operation completed successfully.
NULL	A problem occurred whilst trying to obtain the version information.

### Notes

The `dpci_api_version_string()` function first appeared in the DirectPCI API version 1.2.1.

### See Also

`dpci_api_version()` p.97  
`dpci_drv_version_string()` p.102

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
const char *version_string = dpci_api_version_string();
if (version_string != NULL)
{
    printf("API Version is %s \n", version_string);
```

A further example can be found in the `io_demo.c` program.

## 2.5.3 GET DRIVER VERSION CODE

### Definition

```
#include <dpci_core_api.h>

#define DPCI_VERSION_MAJOR(ver) (((ver) & 0xff000000) >> 24)
#define DPCI_VERSION_MINOR(ver) (((ver) & 0x00ff0000) >> 16)
#define DPCI_VERSION_MICRO(ver) (((ver) & 0x0000ff00) >> 8)
#define DPCI_VERSION_BUILD(ver) ((ver) & 0x000000ff)
int dpci_drv_version(void);
```

### Parameters

None.

### Description

This function returns the version code of the API kernel driver currently loaded on the system. The version code is made of three components: the major number (bits 24-31), the minor number (bits 16-23) and the micro number

This function is provided mainly to aid debugging and distribution management.

### Return Value

- |    |   |
|----|---|
| >0 | The operation completed successfully.                               |
| -1 | A problem occurred whilst trying to obtain the version information. |

### Notes

The version code does not include information relating to the type of release, e.g. alpha, beta, final. See *dpci\_api\_version\_string()*.

The *dpci\_drv\_version()* function first appeared in the DirectPCI API version 1.2.1.

### See Also

- [dpci\\_drv\\_version\\_string\(\)](#) p.102
- [dpci\\_api\\_version\(\)](#) p.97

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
int version_code = dpci_drv_version();
if (version_code != -1)
{
    printf("Driver version is %d.%d.%d.%d\n",
    version_code / 16777216, (version_code / 65536) % 16777216,
    (version_code / 256) % 65536, version_code % 256);
```

```
DPCI_VERSION_MAJOR(version_code),  
DPCI_VERSION_MINOR(version_code),  
DPCI_VERSION_MICRO(version_code),  
DPCI_VERSION_BUILD(version_code));  
}
```

A further example can be found in the io\_demo.c program.

## 2.5.4 GET DRIVER VERSION STRING

### Definition

```
#include <dpci_core_api.h>

const char *dpci_drv_version_string(void);
```

### Parameters

None.

### Description

This function returns the textual version string of the API kernel driver currently loaded on the system. The string returned includes a representation of the type of release, e.g. alpha, beta etc.

This function is provided mainly to aid debugging and distribution management.

### Return Value

>0	The operation completed successfully.
NULL	A problem occurred whilst trying to obtain the version information.

### Notes

The `dpci_api_version_string()` function first appeared in the DirectPCI API version 1.2.1.

### See Also

`dpci_api_version()` p.97  
`dpci_drv_version_string()` p.102

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
const char *version_string = dpci_drv_version_string();
if (version_string != NULL)
{
    printf("Kernel driver version is %s \n", version_string);
```

A further example can be found in the `io_demo.c` program.

## 2.5.5 GET HARDWARE VERSION/REVISION

### Definition

```
#include <dpci_core_api.h>

int dpci_drv_hw_version(int *hwidp, int *revp);
```

### Parameters

hwidp	location to place hardware ID in.
revp	location to place hardware revision in.

### Description

This function returns the hardware device ID and hardware device revision via the two pointers provided by the caller. The hardware ID is equivalent to the 16-bit PCI device id; the hardware revision is equivalent to the 8-bit PCI revision number.

The hardware device ID is placed in the memory pointed to by *hwidp* if that is non-NULL.

The hardware revision is placed in the memory pointed to by *revp* if that is non-NULL.

This function is provided mainly to aid debugging and distribution management.

### Return Value

0	The operation completed successfully.
-1	A problem occurred whilst trying to obtain the version information.

### Notes

The `dpci_drv_hw_version()` function first appeared in the DirectPCI API version 1.2.1.

### See Also

`dpci_api_version()` p.97  
`dpci_drv_version()` p.100

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

int hwdev, rev;
if (dpci_drv_hw_version(&hwdev, &rev) != -1)
{
    printf("Hardware is device %x, revision %d\n", hwdev, rev);
```

A further example can be found in the `io_demo.c` program.

## 2.5.6 GET DRIVER HARDWARE PROFILE

### Definition

```
#include <dpci_core_api.h>

int dpci_drv_hw_profile(struct dpci_hardware_profile *dhwp)
```

### Parameters

**dhwp** Structure to be filled in by Kernel driver

### Description

returns the hardware board identification profile structure.

### Return Value

-1	An error occurred
0	successful

### Notes

The `dpci_drv_hw_profile()` function first appeared in the DirectPCI API version 3.1.1.

### Example

```
#
```

## 2.6 EVENT STREAMING API

The event streaming API is a new feature in DirectPCI v3.0.0. This was added to enhance the programming experience when attempting to capture digital input changes on several input ports and receive information about other sources of events in the system, notably intrusion detection changes.

The event streaming API provides these key characteristics:

- Multiple programs (threads) may each open event streams for the same event source (e.g. a digital input port, the IDLP) and each will safely receive all events without contention with other threads. One event stream per program thread is supported.
- Programs may choose to receive events about all digital input ports, the IDLP, the 80-0062 temperature sensor and the power fail detect sensor from a single call of an API function.
- As events are queued internally for each interested program thread, events are not lost when they occur while a thread is handling an earlier event.
- Digital input edge detection is now enhanced to support automatic edge retriggering to ensure that following edges of the opposite type are automatically enabled following an edge of one type.
- Events are time-stamped so that latency factors in event delivery can be estimated and accounted for.
- It is possible to have the API call a user program back with a specific function and parameter whenever an event occurs.
- A new facility exists to *de-bounce* quick successions of input changes on a digital input port caused by noisy mechanical contacts.

It is recommended that all new programming projects use this API in preference to other means of event delivery.

## 2.6.1 DPCI\_EVENT\_T

### Definition

```
#include <dpci_core_api.h>

typedef struct dpci_event
{
    unsigned int          de_type;
    union
    {
        dpci_io_event_t    dio_event;
        dpci_idlp_event_t idlp_event; // IDLP event
    }                     de_data;
    dpci_timestamp_t     de_ts;      // Timestamp
    unsigned char        de_ts_str[32]; // unused
    dpci_timestamp_t     de_ts_delta; // difference in time since the
                                    // previous event; (or) for the
                                    // first event, time since the
                                    // start of event_wait
} dpci_event_t;

#define idlp_event(epv) (epv)->de_data.idlp_event
#define dio_event(epv)  (epv)->de_data.dio_event

#define EVENT_NONE_ANY  0
#define EVENT_DIG_IP   1
#define EVENT_IDLP     2
#define EVENT_PFD      4
#define EVENT_TS       8
```

### Members

Field	Meaning
de_type	States what type of event this structure describes.
de_data	Provides data pertaining to the specific event type determined by de_type.
de_ts	The time of the event in milliseconds. See <i>Notes</i> below.
de_ts_str[32]	Unused.
de_ts_delta	The period in milliseconds since the previous event.

Figure 32. Members of struct dpci\_event

The *de\_type* determines what kind of event is being described

<b>de_type</b>	<b>Value</b>	<b>Description</b>
EVENT_NONE_ANY	0	No specific type of event is being described; this would never be received from API calls.
EVENT_DIG_IP	1	This event refers to a change to a digital input line. The data in the dio_event member of the de_data union are valid.
EVENT_IDLP	2	This event refers to an IDLP event. The data in the idlp_event member of the de_data union are valid.
EVENT_PFD	4	This event refers to a power-fail detect signal change. There is no event-specific data for this type of event.
EVENT_TS	8	This event refers to a temperature sensor event. There is no event-specific data for this type of event.

Figure 33. Types of event for struct dpci\_event.de\_type

## Notes

The dpci\_event\_t data type appeared in version 3.0.0 of DirectPCI SDK & Run-time. The timestamp reported in de\_ts is represented in millisecs since an epoch. However, the definition of the epoch itself is operating system dependent. On Windows, this value is "the time since midnight on January 1, 1601" whereas on Linux it is "the time since midnight on January 1, 1970".

## Availability

The dpci\_event\_t data type appeared in version 3.0.0 of DirectPCI SDK & Run-time.

## 2.6.2 WAIT FOR EVENTS OF ALL TYPES

### Definition

```
#include <dpci_core_api.h>

int dpci_ev_wait_all(dpci_event_t *eventp, int timeout_ms);
```

### Parameters

eventp	A valid pointer to allocated memory, into which an event will be placed if one was received.
timeout_ms	The maximum time to wait for an event to be returned before giving up.

### Description

This function attempts to read a single event from the calling thread's event queue and return it to the caller. The function will wait a maximum of *timeout\_ms* for an event to become available.

Any event qualifies to be returned by this function and no filtering is performed.

### Return Value

- >0 An event was received.
- 0 No event was received.
- 1 An error occurred and no event was received.

### Notes

Due to the imprecise nature of OS scheduling, the actual time before the function returns may be somewhat longer than the specified timeout depending upon system load induced by other programs and services.

### Availability

This function was added in v3.0.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

dpci_event_t event = {0};
int ret = 0;
char buffer [32] = {0};

while (1)
```

```
{
    ret = dpci_ev_wait_all(&event, WAIT_TIMEOUT_MS);
    if (ret < 0)
    {
        fprintf(stderr,
                "%s: Couldn't wait on DPCI Events: %s\n",
                argv0, dpci_last_os_error_string());
        continue;
    }
    else if (ret == 0)
        continue;

    switch (event.de_type)
    {
    case EVENT_DIG_IP:
        printf("DIGITAL I/P Interrupt\t");
        printf("Change Mask - 0x%08lx, Dig i/p: 0x%08lx\n",
               dio_event(&event).change_mask,
               dio_event(&event).dio_value);
        break;

    case EVENT_IDLP:
        printf("IDLP Event\t");
        printf("[%d] %s %04d/%02d/%02d %02d:%02d:%02d\n",
               idlp_event(&event).eventcode,
               dpci_id_event_name(idlp_event(&event).eventcode),
               idlp_event(&event).year + 2000,
               idlp_event(&event).month,
               idlp_event(&event).day,
               idlp_event(&event).hour,
               idlp_event(&event).min,
               idlp_event(&event).sec);
        break;

    case EVENT_PFD:
        printf("PowerFail Detect Event \n");
        break;

    case EVENT_TS:
        printf("Temperature-Sensor Alarm\n");
        break;

    default:
        printf("Unknown Event\n");
    }
}
```

## 2.6.3 WAIT FOR SPECIFIC EVENT TYPES

### Definition

```
#include <dpci_core_api.h>

int dpci_ev_wait(dpci_event_t *eventp,
                  struct event_mask_t event_mask,
                  int timeout_ms);

struct event_mask_t
{
    unsigned int er_type;
    unsigned long int_mask;
    unsigned long edge_state;
    unsigned long autoconfig_edge;
};
```

### Parameters

<b>eventp</b>	A valid pointer to allocated memory, into which an event will be placed if one was received.
<b>event_mask</b>	A structure specifying which types of events to match for return.
<b>timeout_ms</b>	The maximum time to wait for an event to be returned before giving up.

### Description

This function attempts to read a single event from the calling thread's event queue and return it to the caller. The function will wait a maximum of *timeout\_ms* for an event to become available.

Events must match the *event\_mask*. Its fields are defined thus:

Figure 34. Members of struct *event\_mask\_t*

<b>Member</b>	<b>Description</b>
<b>er_type</b>	The type of event. See Figure 32: Members of struct <i>dpci_event</i> on p106
<b>int_mask</b>	For cases where <i>er_type</i> == <i>EVENT_DIG_IP</i> , the mask of digital input lines for which events should be reported.

Member	Description
edge_state	For cases where er_type == EVENT_DIG_IP, this bit-mask determines whether the next event for a given digital input is for a 0-1 transition or 1-0 transition. A 1 indicates report is desired for a 0 to 1 transition.
autoconfig_edge	For cases where er_type == EVENT_DIG_IP, this bit-mask indicates which digital input lines should automatically be reconfigured to report events for the opposite edge type once a change has been detected. The device driver will automatically reconfigure the digital input accordingly.

## Return Value

- >0 An event was received.
- 0 No event was received.
- 1 An error occurred and no event was received.

## Notes

Due to the imprecise nature of OS scheduling, the actual time before the function returns may be somewhat longer than the specified timeout depending upon system load induced by other programs and services.

Waiting for events qualified by the EVENT\_IDLP bit precludes also using dpci\_id\_readevent(). Only one caller can reliably read new events from the IDLP.

## Availability

This function was added in v3.0.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

dpci_event_t event = {0};
struct event_mask_t event_mask = {0};
int ret = 0;
char buffer [32] = {0};

.

.

// look for IDLP and digio port 0 events.
event_mask.er_type = EVENT_IDLP | EVENT_DIG_IP;
event_mask.int_mask = 0x000000FF;           // port 0 only
event_mask.edge_state = 0x00000000;        // interrupt on 1->0 transition
event_mask.autoconfig_edge = 0x000000FF; // auto-reconfigure edgestate

while (1)
```

```
{  
    ret = dpci_ev_wait(&event, event_mask, WAIT_TIMEOUT_MS);  
    if (ret < 0)  
    {  
        fprintf(stderr,  
                "%s: Couldn't Wait on DPCI Events: %s\n",  
                argv0, dpci_last_os_error_string());  
        continue;  
    }  
    else if (ret == 0)  
        continue;  
  
    switch (event.de_type)  
    {  
        case EVENT_DIG_IP:  
            printf("DIGITAL I/P Interrupt\t");  
            printf("Change Mask - 0x%08lx, Dig i/p: 0x%08lx\n",  
                   dio_event(&event).change_mask,  
                   dio_event(&event).dio_value);  
            break;  
        case EVENT_IDLP:  
            printf("IDLP Event\t");  
            printf("[%d]%s %04d/%02d/%02d %02d:%02d:%02d\n",  
                   idlp_event(&event).eventcode,  
                   dpci_id_event_name(idlp_event(&event).eventcode),  
                   idlp_event(&event).year + 2000,  
                   idlp_event(&event).month,  
                   idlp_event(&event).day,  
                   idlp_event(&event).hour,  
                   idlp_event(&event).min,  
                   idlp_event(&event).sec);  
            break;  
        default:  
            printf("Unknown Event\n");  
            break;  
    }  
}
```

## 2.6.4 REGISTER A CALLBACK

### Definition

```
#include <dpci_core_api.h>

typedef void (*dpci_ev_callback_t)(struct dpci_event *evp, void *data);

int dpci_ev_register_callback(dpci_ev_callback_t func,
                             void *data,
                             struct dpci_event *evp);
```

### Parameters

func	A function to be called when an event matching <i>evp</i> is received
data	A token to be passed to the callback function when the event occurs.
evp	A specification for the type of event to match.

### Description

Dpci\_ev\_register\_callback arranges for the function *func* to be called automatically when an event occurs that matches the contents of *evp*.

When a matching event is found, the callback is invoked with the *data* argument and a pointer to the event which has just occurred.

Event matching occurs as follows:

- At least one bit must be set in the *de\_type* member of *evp*;
- If the value of *de\_type* is *EVENT\_DIG\_IP* then the contents of *evp->de\_data* are used to filter events before calling the callback;
- If multiple bits are set in *de\_type* then the *de\_data* union's fields are not used and all events of the types specified in *de\_type* are matched.

For digital input events the *dio\_value* and *change\_mask* values work thus:

- If both *dio\_value* and *change\_mask* are zero, then all digital input events are matched;
- If *change\_mask* is non-zero, then digital input events are matched where one or more digital inputs has changed whose bit is also set in *change\_mask* and where the new state of that input line is the same as the state of the corresponding bit in *dio\_value*.
- If *change\_mask* is zero and *dio\_value* is non-zero, then digital input events are matched where one or more digital inputs has changed whose bit is also set in *dio\_value*; the actual line level of that bit is ignored in this case.

For IDLP events, the `eventcode` member is matched against the incoming event if it is non-zero; otherwise, all IDLP events are matched. It is not possible to match a selection of IDLP events with a single callback.

## Return Value

- |    |                                   |
|----|-----------------------------------|
| 0  | Callback registered successfully. |
| -1 | Callback not registered.          |

## Notes

It is possible to register the same callback multiple times; in this event, the callback will be called as many times as it has been registered. The same function and data parameters may be used as many times as required.

It is not presently possible to match individual IDLP events or a subset of all possible events.

It is not permitted to register a callback from within a callback. Application behaviour is undefined for this case.

It is not supported to perform any event reception functions from within a callback. Application behaviour is undefined for this case.

This function is thread-safe.

## Availability

`Dpci_ev_register_callback` first appeared in DirectPCI v3.0.0

## Example

An example can be found in the `callback_demo.c` program.

## 2.6.5 UNREGISTER A CALLBACK

### Definition

```
#include <dpci_core_api.h>

typedef void (*dpci_ev_callback_t)(struct dpci_event *evp, void *data);

int dpci_ev_unregister_callback(dpci_ev_callback_t func,
                                void *data,
                                struct dpci_event *evp);
```

### Parameters

- func      A function passed to *dpci\_ev\_register\_callback*
- data      A token passed to *dpci\_ev\_register\_callback*
- evp      A specification for the type of event passed to *dpci\_ev\_register\_callback*

### Description

This function removes from the callback list a callback that was previously installed using [dpci\\_ev\\_register\\_callback\(\)](#).

### Return Value

- 0      Callback unregistered successfully.
- 1      Callback not registered.

### Notes

As it is possible to register the same callback multiple times, the callback must be removed as many times before all callbacks cease.

It is not permitted to unregister a callback from within a callback. Application behaviour is undefined for this case.

This function is thread-safe.

### Availability

*Dpci\_ev\_unregister\_callback* first appeared in DirectPCI v3.0.0

### Example

An example can be found in the *callback\_demo.c* program.

## 2.6.6 SET INPUT DE-BOUNCE PARAMETERS

### Definition

```
#include <dpci_core_api.h>

int dpci_ev_set_debounce(unsigned long mask, int ms);
```

### Parameters

mask	The bit mask of digital input lines to be de-bounce.
ms	The de-bounce period in milliseconds.

### Description

The *dpci\_ev\_set\_debounce* function allows multiple input events caused by a mechanical switch to be effectively “de-bounced” into a single event.

Any digital input line for which there is a 1 bit set in the corresponding position in *mask* is de-bounced so that no more than 1 event will be reported in the time period determined by *ms*.

This interface only de-bounces events to be reported via the callback feature. When using *dpci\_ev\_wait()* or *dpci\_ev\_wait\_all()*, all events are still received without de-bounce processing.

### Return Value

0	Debounce parameters set successfully.
-1	Failed to set debounce parameters.

### Notes

If, in the time period *ms*, a de-bounced digital input changes to its opposite state and then reverts to its original state then a single event will be the result which confirms the line in its original state.

By default no de-bounce function operates.

### See Also

*dpci\_ev\_register\_callback()* - p113  
*dpci\_ev\_unregister\_callback()* - p115

### Availability

*dpci\_ev\_register\_callback* first appeared in DirectPCI v3.0.0

## **Example**

An example can be found in the *callback\_demo.c* program.

## 2.7 HOST WATCHDOG API

### 2.7.1 ENABLE IDLP Host WATCHDOG

#### Definition

```
#include <dpci_core_api.h>

int dpci_wd_enable(int interval_seconds);
```

#### Parameters

interval_seconds	The maximal interval in seconds the watch-dog should wait for a watch-dog reset command before rebooting the system.
------------------	--

#### Description

This function enables the watch-dog capability embedded in the intrusion detection processor. When enabled, the watch-dog waits a specified number of seconds and then reboots the main CPU if a watch-dog reset command has not been received. The watch-dog timer can be reset to its programmed interval by calling `dpci_wd_reset()`.

The allowed range of time-outs differs according to the IDLP firmware version:

- For IDLP firmware v57 and newer, the range is 1 – 1024 seconds;
- For older firmware, the range 1 – 255 seconds.

Outside the range, the value -1 is returned to signal an error condition.

Care should be taken when adding watch-dog functionality into programs. If the watch-dog is being reset by a thread of a program, then any interruption to that program might cause an inadvertent system reboot. This can happen also when the program is being debugged.

#### Return Value

0	Watchdog enabled successfully.
-1	Watchdog could not be enabled.

#### Notes

The watch-dog is enabled automatically when the system is reset or powered on. On ConnectBus-II® using IO boards 80-1003 and 80-0062, a separate watchdog is available for the ioboard. See 2.17.4 *Enable I/O Board Watchdog* to enable io-board watchdog.

#### Availability

The `dpci_wd_enable()` function first appeared in DirectPCI v1.0.0.

## **Example**

See example of Watchdog Reset (p.121). A further example exists in the *id\_demo.c* program.

## 2.7.2 DISABLE IDLP Host WATCHDOG

### Definition

```
#include <dpci_core_api.h>

int dpci_wd_disable(void);
```

### Parameters

None

### Description

This function stops the watchdog function on the intrusion detection processor from resetting the system. It disables all watchdog timer activity until a:

- another call to dpci\_wd\_enable()
- a system reset.

### Return Value

- |    |                                 |
|----|---------------------------------|
| 0  | Watchdog disabled successfully. |
| -1 | Watchdog could not be disabled. |

### Notes

On ConnectBus-II® using IO boards 80-1003 and 80-0062, a separate watchdog is available for the ioboard. See 2.17.5 *Disable I/O Board Watchdog* to disable io-board watchdog.

### Availability

The dpci\_wd\_disable() function first appeared in DirectPCI v1.0.0.

### Example

See example of Watchdog Reset (p.121).

A further example can be found in the *id\_demo.c* program.

## 2.7.3 DPCI WATCHDOG FUNCTIONS: WATCHDOG RESET

### Definition

```
#include <dpci_core_api.h>

int dpci_wd_reset(void);
```

### Parameters

None

### Description

Resets the watchdog timer to start counting again down from the maximum interval specified by a previous call to `dpci_wd_enable()`.

### Return Value

- |    |                              |
|----|------------------------------|
| 0  | Watchdog reset successfully. |
| -1 | Watchdog could not be reset. |

### Notes

On ConnectBus-II® using IO boards 80-1003 and 80-0062, a separate watchdog is available for the ioboard. See 2.17.6 *Pat I/O Board Watchdog* to reset io-board watchdog.

### Availability

The `dpci_wd_reset()` function first appeared in DirectPCI v1.0.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

void stopwdog(int signo)
{
    dpci_wd_disable();
    exit(signo);
}

if (fork() == 0) {
    if (dpci_wd_enable(60))
    {
        printf("ERROR: Watchdog could not be enabled!\n");
        exit (255);
    }

    /*
     * Make sure we stop the watchdog if we get a signal;
     * we probably want to do this for other signals as
     * well as SIGTERM.
     */
}
```

```
signal(SIGTERM, stopwdog);

while (1) {
    /*
     * Only sleep for half-period due to latencies
     */
    dpci_wd_reset();
    sleep(WATCHDOG_TIMEOUT / 2);
}
```

A further example can be found in the id\_demo.c program.

## 2.8 INTRUSION DETECTION AND LOGGING PROCESSOR API

### 2.8.1 ERROR CODES COMMON TO ALL IDLP API ENTRIES

The following error codes are common to allow calls supported by the IDLP API.

Figure 35. Error Codes Common to All IDLP API Calls

Error Code	Explanation
Linux:ETTY Windows: ERROR_NOT_SUPPORTED	This command is not supported on the current IDLP version/implementation: <ul style="list-style-type: none"><li>• The IDLP version on the board is too old to support the command;</li><li>• The IDLP version is newer and the given command has been deprecated.</li><li>• The command is only supported on some builds of the IDLP, i.e. on one DPX-series board but not the current one.</li></ul>
Linux:EINVAL Windows: ERROR_INVALID_PARAMETER	The parameter given to the command is invalid.
Linux:EIO Windows: ERROR_IO_DEVICE	The command failed due to an unexpected error although this command would ordinarily be expected to succeed.  Note that the dpci_core driver will normally retry a failed command 3 times before giving up completely and returning an error code to the caller.  The caller may still retry the command as needed.

Error code returns from the IDLP are more rigorously relayed and enforced in v60 firmware and dpci\_core driver v3.1.0.1069 and later than in previous releases of firmware and driver. In previous releases, most IDLP errors are relayed solely as an I/O error (linux EIO, Windows ERROR\_IO\_DEVICE).

## 2.8.2 GET IDLP FIRMWARE VERSION

### Definition

```
#include <dpci_core_api.h>

int dpci_id_fwversion(void);
```

### Parameters

None

### Description

This function returns the major part of the firmware version from the intrusion detection and logging processor (IDLP). The version number is returned in signed binary format. The version numbering of the intrusion detection and logging processor's firmware starts at 1 and increases monotonically.

### Return Value

- |    |  |
|----|--|
| >0 | The integer PIC firmware version number                                  |
| -1 | An error occurred. See 2.8.1 for an explanation of possible error codes. |

### Note

Until version 56 firmware, there were no major or minor version number components encoded in the version number. Starting from v56, a build number has been added in addition to two configuration bytes. For these versions of firmware, this function only returns the major part of the version number.

### Availability

The `dpci_id_fwversion()` function first appeared in DirectPCI v1.0.0.

Firmware support exists from v1 and later.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

int fwversion = dpci_id_fwversion();

if (fwversion == -1)
{
    fprintf(stderr, "Unable to get firmware version\n");
    exit(1);
}
printf("Firmware version %d.\n", fwversion);
```

A further example can be found in the `id_demo.c` program.

## 2.8.3 GET FULL IDLP FIRMWARE VERSION

### Definition

```
#include <dpci_core_api.h>
#include <dpci_core_hw.h>

#define IDLP_VERSION_MAJOR(verfull)      (((verfull) & 0xff)
#define IDLP_VERSION_BUILD(verfull)       (((verfull) >> 8) & 0xff)
#define IDLP_VERSION_CONF1(verfull)        (((verfull) >> 16) & 0xff)
#define IDLP_VERSION_CONF2(verfull)        (((verfull) >> 24) & 0xff)

int dpci_id_fwversion_full(void);
```

### Parameters

None

### Description

This function returns the full firmware version from the intrusion detection and logging processor (IDLP). The major version number of the intrusion detection and logging processor's firmware starts at 1 and increases monotonically.

The four fields encoded in the full firmware version number returned are as show:

Figure 36. IDLP Firmware Version Number Components

Macro	Meaning
IDLP_VERSION_MAJOR	The Major version number (10, 32, 48, 56 etc.)
IDLP_VERSION_BUILD	The build number. This is only present from v56 firmware.
IDLP_VERSION_CONF1	Configuration word 1. This component is reserved for future use and will ordinarily be zero.
IDLP_VERSION_CONF2	Configuration word 2. This word indicates aspects of the build.  Bit 0 set: debug mode enabled. Bit 1 set: internal development build. Bit 3 set: single Li+ battery supported. Bit 6 set: access to IDPROM not implemented. Bit 7 set: intrusion inputs 6 and 7 implemented.  All other bits are unused or reserved for internal development uses and values should be ignored.

## Return Value

- |    |  |
|----|--|
| >0 | The integer PIC firmware version number                                  |
| -1 | An error occurred. See 2.8.1 for an explanation of possible error codes. |

## Notes

Until version 56 firmware, there were no build number or other components encoded in the version number. For these versions of firmware, this function only returns the major part of the version number. Starting from v56, a build number and two configuration bytes has been included in addition to the major number.

## Availability

The `dpci_id_fwversionfull()` function first appeared in DirectPCI v3.0.0.

Firmware support exists from v56 and later.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>
#include <dpci_core_hw.h>

...
int fwversion = dpci_id_fwversion_full();
if (fwversion == -1)
{
    fprintf(stderr, "Unable to get firmware version\n");
    exit(1);
}
printf("IDLP firmware version: %02d.%d (%d.%d)\n",
       IDLP_VERSION_MAJOR(fwversion),
       IDLP_VERSION_BUILD(fwversion),
       IDLP_VERSION_CONF1(fwversion),
       IDLP_VERSION_CONF2(fwversion));
```

## 2.8.4 GET IDLP INTRUSION LINE STATUS

### Definitions

```
#include <dpci_core_api.h>
#include <dpci_core_hw.h>

int dpci_id_intrusionstatus(void);
#define INTRUSION0_MASK 0x01
#define INTRUSION1_MASK 0x02
#define INTRUSION2_MASK 0x04
#define INTRUSION3_MASK 0x08
#define INTRUSION4_MASK 0x10
#define INTRUSION5_MASK 0x20
#define INTRUSION6_MASK 0x40
#define INTRUSION7_MASK 0x80
```

### Parameters

None

### Description

This function returns a bit mask indicating the state of each intrusion line. A bit is set in the mask if the corresponding intrusion input is shorted to ground, i.e. closed circuit.

### Return Value

- |     |   |
|-----|---|
| -1  | The intrusion status could not be read. See 2.8.1 for an explanation of possible error codes. |
| >=0 | The intrusion status bit-mask.  |

### Notes

This function does not cause the IDLP to issue an immediate read of the intrusion lines; instead, the IDLP returns the status obtained at the last scheduled check of the intrusion lines.

This function is only available on the DPX-116 and newer boards where the IDLP firmware revision is 24 or greater. This function is not supported on the DPX-114 or DPX-115 boards.

### Availability

The `dpci_id_intrusionstatus()` function first appeared in the DirectPCI API version 1.2.1.

Firmware support exists from v24 and later.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
mask = dpci_id_intrusionstatus();
if (mask == -1)
{
    perror("dpci_id_intrusionstatus failed");
}
printf("Intrusion detection circuit#0 is %s\n",
       (status & INTRUSION0_MASK) ? "closed" : "open");

printf("Intrusion detection circuit#1 is %s\n",
       (status & INTRUSION1_MASK) ? "closed" : "open");

printf("Intrusion detection circuit#2 is %s\n",
       (status & INTRUSION2_MASK) ? "closed" : "open");

printf("Intrusion detection circuit#3 is %s\n",
       (status & INTRUSION3_MASK) ? "closed" : "open");

printf("Intrusion detection circuit#4 is %s\n",
       (status & INTRUSION4_MASK) ? "closed" : "open");

printf("Intrusion detection circuit#5 is %s\n",
       (status & INTRUSION5_MASK) ? "closed" : "open");

printf("Intrusion detection circuit#6 is %s\n",
       (status & INTRUSION6_MASK) ? "closed" : "open");

printf("Intrusion detection circuit#7 is %s\n",
       (status & INTRUSION7_MASK) ? "closed" : "open");
```

A further example can be found in the *id\_demo.c* program.

## 2.8.5 SET IDLP SLEEP PERIOD

### Definitions

```
#include <dpci_core_api.h>

int dpci_id_setsleeptime(int sleepcode);
```

### Parameters

**sleepcode** The code for the chosen sleep time.

### Description

Sets the minimum period between which the intrusion detection and logging processor will check the states of the intrusion detection lines while the board is powered down. The following table enumerates all the allowable sleep periods and the value of *sleepcode* which corresponds to each

<b>C macro for sleepcode</b>	<b>Value</b>	<b>Period</b>
INTRUS_SLEEP_1SEC	0x31	1 second
INTRUS_SLEEP_500MSEC	0x32	500 ms
INTRUS_SLEEP_250MSEC	0x33	250 ms
INTRUS_SLEEP_125MSEC	0x34	125 ms

Figure 37. IDLP Firmware Sleep-time Settings

### Return Value

- 0 Success
- 1 An error occurred. See 2.8.1 for an explanation of possible error codes.

### Notes

The IDLP checks the intrusion circuits every 125ms while the computer is powered up.

The default period for checking while powered down is 1 second. It is not normally advisable to change this period without good reason. While powered down, the IDLP operates from battery power if no regular power supply is functioning. Increasing the frequency of IDLP checks therefore will considerably shorten battery life.

### Availability

The `dpci_id_setsleeptime()` function first appeared in the DirectPCI API version 1.2.1.

IDLP f/w v60.18 and later rejects all attempts to set the sleep time with any code other than `INTRUS_SLEEP_1SEC`.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
if (dpci_id_setsleeptime(INTRUS_SLEEP_1SEC) == -1)
{
    perror("dpci_id_setsleeptime()");
}
```

A further example can be found in the *id\_demo.c* program.

## 2.8.6 GET NUMBER OF UNREAD IDLP EVENTS

### Definitions

```
#include <dpci_core_api.h>

int dpci_id_numevents(void);
```

### Parameters

None

### Description

The function returns the number of events logged by the intrusion detection and logging processor. Most events types represent intrusion events although other event types are also logged; see page 132 for further details.

### Return Value

>=0	The number of intrusions detected that can then be read
-1	An error occurred. See 2.8.1 for an explanation of possible error codes.

### Availability

The `dpci_id_numevents()` function first appeared in the DirectPCI API version 1.2.0.

Firmware support exists from v1 and later.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
NumOfEvents = dpci_id_numevents();
if (NumOfEvents == -1)
{
    perror("dpci_id_numevents");
}
```

A further example can be found in the `id_demo.c` program.

## 2.8.7 READ AN IDLP EVENT

### Definition

```
#include <dpci_core_hw.h>
#include <dpci_core_api.h>

struct idevent
{
    char eventcode;
    char year;      /* since 2000          */
    char month;     /* 1 = Jan             */
    char day;       /* 1 = 1st of month    */
    char hour;
    char min;
    char sec;
};

int dpci_id_readevent(struct idevent *inevent);
```

### Parameters

inevent	A structure for the return information on each intrusion event to populate
---------	--

### Description

Reads the most recent unread intrusion event (including time and date information) from the intrusion detection and logging processor. The event is placed in the structure pointed to by *inevent*.

The event structure struct idevent has the following fields:

Figure 38. Members of struct idevent

Member	Definition
eventcode	This is the number of an event reported by the IDLP. A setting of 0 is used where no event occurred or when setting or obtaining the date and time.
year	This is the number of years since 2000.
month	This is the month number; January = 1
day	This is the day number: 1 <sup>st</sup> of the month = 1
hour	This is the hour number in 24-hour clock format, i.e. 0..23.
min	This is the number of minutes, 0..59

Member	Definition
sec	This is the number of seconds, 0..59.

Add 2000 to the *year* field to obtain the year of the event. The month and day fields are both 1-based, i.e. month 1 means January, day 1 means the 1<sup>st</sup> day of the month.

The event codes reported by the logging processor are defined in *dpci\_core\_hw.h* as follows:

Figure 39. IDLP Event Numbers

#define ID_EVENT_NONE	0
#define ID_EVENT_INTRUS0_CLOSED_CIRCUIT	1
#define ID_EVENT_INTRUS1_CLOSED_CIRCUIT	2
#define ID_EVENT_INTRUS2_CLOSED_CIRCUIT	3
#define ID_EVENT_INTRUS3_CLOSED_CIRCUIT	4
#define ID_EVENT_SYSRESET_10	5
#define ID_EVENT_SYSRESET_01	6
#define ID_EVENT_WDRESET	7
#define ID_EVENT_INTERNAL_RESET_POWERUP	8
#define ID_EVENT_INTRUS0_OPEN_CIRCUIT	9
#define ID_EVENT_INTRUS1_OPEN_CIRCUIT	10
#define ID_EVENT_INTRUS2_OPEN_CIRCUIT	11
#define ID_EVENT_INTRUS3_OPEN_CIRCUIT	12
#define ID_EVENT_INTRUS4_OPEN_CIRCUIT	13
#define ID_EVENT_INTRUS5_OPEN_CIRCUIT	14
#define ID_EVENT_INTRUS4_CLOSED_CIRCUIT	15
#define ID_EVENT_INTRUS5_CLOSED_CIRCUIT	16
#define ID_EVENT_OLDTIME	17
#define ID_EVENT_NEWTIME	18
#define ID_EVENT_INTERNALRESET_WDT	19
#define ID_EVENT_INTERNALRESET_MCLR	20
#define ID_EVENT_DIAGENABLED	21
#define ID_EVENT_BATT1TOOLOW	22
#define ID_EVENT_BATT1CORRECT	23
#define ID_EVENT_BATT2TOOLOW	24
#define ID_EVENT_BATT2CORRECT	25
#define ID_EVENT_BATT3TOOLOW	26
#define ID_EVENT_BATT3CORRECT	27
#define ID_EVENT_IDLPRESETNOEE	28
#define ID_EVENT_INTRUS6_OPEN_CIRCUIT	29
#define ID_EVENT_INTRUS7_OPEN_CIRCUIT	30
#define ID_EVENT_INTRUS6_CLOSED_CIRCUIT	31
#define ID_EVENT_INTRUS7_CLOSED_CIRCUIT	32
#define ID_EVENT_INTERNALRESET_BOR	33
#define MAX_ID_EVENT	33

The section on DirectPCI hardware in the manual for your Advantech Innocore main board explains the meanings of each event type.

The following event codes deserve further explanation:

Figure 40. Noteworthy IDLP Events

Member	Definition
--------	------------

Member	Definition
<code>ID_EVENT_SYSRESET_10</code>	<p>The IDLP detected a high-to-low transition on the system reset line. This can occur for the following reasons:</p> <ul style="list-style-type: none"> <li>• Somebody pressed the reset button.</li> <li>• A soft reset occurred initiated by the OS (or [CTRL]+[ALT]+[DEL]).</li> <li>• The system was powered down.</li> </ul>
<code>ID_EVENT_SYSRESET_01</code>	<p>The IDLP detected a high-to-low transition on the system reset line. This can occur for the following reasons:</p> <ul style="list-style-type: none"> <li>• Somebody pressed the reset button.</li> <li>• A soft reset occurred initiated by the OS (or [CTRL]+[ALT]+[DEL]).</li> </ul> <p>The system powered up.</p>
<code>ID_EVENT_WDRESET</code>	<p>The IDLP's host watch-dog timer expired and so the IDLP attempted to reset the host. Note that this event will still be logged even if the jumper is not fitted which enables the actual reset to take effect.</p>
<code>ID_EVENT_INTERNAL_RESET_POWERUP</code>	<p>The IDLP powered up. Normally this would only be seen if the IDLP powered up because the board is powered down but a new battery was inserted, or if no battery is present when the board is powered on.</p>
<code>ID_EVENT_OLDTIME</code>	<p>This event is logged to record the time and date at which a request to change the time was granted.</p>
<code>ID_EVENT_NEWTIME</code>	<p>This event is logged to record the new time and date after a request to change the time was granted.</p>
<code>ID_EVENT_INTERNAL_RESET_WDT</code>	<p>The IDLP suffered an internal reset triggered by its internal watch-dog timer. This is caused ordinarily by a firmware defect. Please report such occurrences to your Advantech support team contact or your account manager.</p>

<b>Member</b>	<b>Definition</b>
ID_EVENT_INTERNAL_RESET_MCLR	The IDLP suffered an internal reset triggered by the external reset pin on the IDLP chip. This type of reset occurs after the device has been reprogrammed but may also occur in other circumstances; please report such occurrences to your Advantech support team contact or your account manager.
ID_EVENT_IDLPPRESETNOEE	This is caused when the IDLP is unable to detect the storage device for its regular event log. In this event, 48 events may be stored inside the IDLP's internal RAM. Please report such occurrences to your Advantech support team contact or your account manager.

## Return Value

- |    |  |
|----|--|
| 0  | Successfully read an event and populated structure |
| -1 | Failure populating structure.                      |

## Notes

This function will almost always return an event with eventcode set to ID\_EVENT\_NONE if it is called when a user on the system is waiting for IDLP events using the `dpci_id_wait_event()`, `dpci_ev_wait()` or `dpci_ev_wait_all()` APIs. If an event is returned successfully under these circumstances then users of the `dpci_ev_wait()` and `dpci_id_wait()` APIs will not see that event.

## Availability

The function prototype for `dpci_id_readevent()` was unified between Linux and Windows releases at v1.2.0.

Version v1.3.0 saw the introduction of the new and old time events.

Version v2.0.0 saw the introduction of events for battery monitoring and diagnostics mode.

Firmware support exists from v1 and later.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

struct idevent event;

while (1)
{
    /*
     * Make sure we can actually read an event.
     */
    if (dpci_id_readevent(&event) == -1)
    {
        fprintf(stderr, "Failed to read event.\n");
    }

    /*
     * Event code 0 means no further events in the FIFO.
     */
    if (inevent.eventcode == 0)
        break;
    printf("Event code: %d\n", inevent->eventcode);
    printf("Year: %d\n", inevent->year + 2000);
    printf("Month: %d\n", inevent->month);
    printf("Day: %d\n", inevent->day);
    printf("Hour: %d\n", inevent->hour);
    printf("Min: %d\n", inevent->min);
    printf("Sec: %d\n", inevent->sec);
}
```

A further example can be found in the *id\_demo.c* program.

## 2.8.8 READ AN OLD IDLP EVENT

### Definition

```
#include <dpci_core_hw.h>
#include <dpci_core_api.h>

int dpci_id_select_event(int which, struct idevent *inevent);
```

### Parameters

which	The index of the event, where 0 is the most recent
inevent	A structure for the return information on each intrusion event to populate

### Description

Reads the  $n^{\text{th}}$  available previously-read intrusion event from the most recently recorded (which = 0) through to the oldest (which = 254), including time and date information from the intrusion detection and logging processor. The event is placed in the structure pointed to by *inevent*.

Because the IDLP's event log is a fixed size, if there are any events unread in the IDLP's event queue, this number will limit the maximum number of old events that may be read. An attempt to read events beyond the log's end (or not taking into account new or unread events) while result in an "invalid parameter" error (see Figure 35 on p123).

The data type `struct idevent` is detailed in Figure 38: Members of `struct idevent` on p132.

### Return Value

0	Successfully read an event and populated structure
-1	A failure occurred reading the event. See 2.8.1 for an explanation of possible error codes.

### Notes

The function prototype for `dpci_id_readevent()` was unified between Linux and Windows releases at v1.2.0.

With firmware prior to v60 and or a DirectPCI dpci-core driver prior to v3.1.0.1169, the error EIO (Linux) or ERROR\_IO\_DEVICE (Windows) is returned instead of the error code indicated in the description above.

### Availability

The `dpci_id_select_event()` function was introduced in v2.0.0.

Firmware support exists from v56.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
struct idevent oldevent;
int eventno;

for (eventno = 0; eventcode <= 254; eventno++)
{
    if (dpci_id_select_event(eventno, &oldevent) == -1)
    {
#ifdef WIN32
        if (GetLastError() == ERROR_INVALID_PARAMETER ||
            GetLastError() == ERROR_IO_DEVICE)
#else
        if (errno == EINVAL || errno == EIO)
#endif
        {
            printf("New unread events have been logged.\n");
            break;
        }
        perror("Error getting RTC Date and Time");
        break;
    }
    printf("Event code %d last occurred %04d/%02d/%02d "
           "%02d:%02d:%02d\n",
           oldevent.year + 2000,
           oldevent.month,
           oldevent.day,
           oldevent.hour,
           oldevent.min,
           oldevent.sec);
}
```

A further example can be found in the *id\_demo.c* program.

## 2.8.9 WAIT FOR AN IDLP EVENT

### Definition

```
#include <dpci_core_api.h>

int dpci_id_wait_event(struct idevent *inevent,
                       unsigned long timeout_ms);
```

### Parameters

- |            |  |
|------------|--|
| inevent    | A pointer to the user allocated structure of type idevent.                             |
| timeout_ms | The time in milliseconds (ms) the function should wait for an event interrupt to occur |

### Description

*dpci\_id\_wait\_event()* blocks the current thread and waits up to *timeout\_ms* milliseconds for an event to be reported by the intrusion detection and logging processor. Should an event occur during the specified period then it is read and placed in *inevent*. The function returns if either an interrupt or timeout occurred.

The data type `struct idevent` is detailed in Figure 38: Members of struct idevent on p132.

If an event was recorded by the logging processor then the function returns the event details in the idevent structure pointed to by the *inevent* parameter.

The timeout parameter *timeout\_ms* is counted in milliseconds. A setting of 0 causes the function to return almost immediately. A setting of -1 means  $2^{32}-1$  ms – an effective timeout of 4.29 million seconds – 49.7 days. While this might be considered to be near-infinite for some applications, applications should code for the possibility that a time-out condition might still occur.

### Return Value

- |    |   |
|----|---|
| 1  | The function completed successfully and the valid event data was stored in the memory pointed by <i>inevent</i> parameters. |
| 0  | Operation timed-out waiting for an interrupt.   |
| -1 | A failure occurred while waiting for the interrupt. See 2.8.1 for an explanation of possible error codes.                   |

### Notes

It is recommended that this function not be used and that the Event Streaming API (p97) be utilized instead for all event delivery requirements.

If *dpci\_id\_wait\_event()* returns successfully then it is suggested that all outstanding events be read immediately; because the intrusion detection and logging processor is much lower powered than the host CPU it will not always flag an outstanding event immediately upon calling with the result that a delay will be observed between reporting of successive events.

Also, there is an initial latency (some 10ms) occurred when the intrusion detection and logging processor is instructed to enable interrupt delivery. As such, a time-out setting of zero milliseconds will always likely always return 0.

The *dpci\_id\_wait\_event()* function is now considered deprecated as it has been superceded by the *dpci\_ev\_wait()* and *dpci\_ev\_waitall()* functions.

## Availability

The *dpci\_id\_wait\_event()* function first appeared in the DirectPCI API version 1.2.1.

Firmware support exists from v1.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

#define EVENT_TIMEOUT_MS 30000
struct idevent event;

while (1)
{
    switch(dpci_id_wait_event(&event, EVENT_TIMEOUT_MS))
    {
        case 0:
            printf("Time-out!\n");
            continue;
        case -1:
            perror("dpci_id_wait_event failed");
            return;
    }

    do
    {
        /*
         * An event with code 0 means there are no further
         * events in the FIFO.
         */
        if (inevent.eventcode == 0)
        {
            break;
        }
        printf("Event code: %d\n", inevent->eventcode);
        printf("Year: %d\n", inevent->year + 2000);
        printf("Month: %d\n", inevent->month);
        printf("Day: %d\n", inevent->day);
        printf("Hour: %d\n", inevent->hour);
        printf("Min: %d\n", inevent->min);
        printf("Sec: %d\n", inevent->sec);

        /*
         * Attempt to read the next event.
         */
        if (dpci_id_readevent(&event) == -1)
    {
```

```
    perror("Failed to read event");
    break;
}
} while (inevent.eventcode != ID_EVENT_NONE);
```

A further example can be found in the *id\_demo.c* program.

## 2.8.10 GET NAME FOR IDLP EVENT NUMBER

### Definitions

```
#include <dpci_core_api.h>

const char *dpci_id_event_name(int eventcode);
```

### Parameters

eventcode	code for the event whose name is required.
-----------	--

### Description

This function returns a text string which describes the event whose code is passed in *eventcode*. The valid event codes are listed on page 132.

### Return Value

The return value is a text string for the given event code. The following table summarises the event codes and the corresponding return values.

Figure 41. IDLP Event Macros and Corresponding Text

Event code macro	String
ID_EVENT_NONE	“no event”
ID_EVENT_INTRUS0_CLOSED_CIRCUIT	“intrusion line#0 c/c”
ID_EVENT_INTRUS1_CLOSED_CIRCUIT	“intrusion line#1 c/c”
ID_EVENT_INTRUS2_CLOSED_CIRCUIT	“intrusion line#2 c/c”
ID_EVENT_INTRUS3_CLOSED_CIRCUIT	“intrusion line#3 c/c”
ID_EVENT_SYSRESET_10	“system-reset 1->0”
ID_EVENT_SYSRESET_01	“system-reset 0->1”
ID_EVENT_WDRESET	“watchdog reset host”
ID_EVENT_INTERNALRESET_POWERUP	“IDLP internal reset: power up”
ID_EVENT_INTRUS0_OPEN_CIRCUIT	“intrusion line#0 o/c”
ID_EVENT_INTRUS1_OPEN_CIRCUIT	“intrusion line#1 o/c”
ID_EVENT_INTRUS2_OPEN_CIRCUIT	“intrusion line#2 o/c”
ID_EVENT_INTRUS3_OPEN_CIRCUIT	“intrusion line#3 o/c”
ID_EVENT_INTRUS4_OPEN_CIRCUIT	“intrusion line#4 o/c”

Event code macro	String
ID_EVENT_INTRUS5_OPEN_CIRCUIT	“intrusion line#5 o/c”
ID_EVENT_INTRUS4_CLOSED_CIRCUIT	“intrusion line#4 c/c”
ID_EVENT_INTRUS5_CLOSED_CIRCUIT	“intrusion line#5 c/c”
ID_EVENT_INTRUS6_OPEN_CIRCUIT	“intrusion line#6 o/c”
ID_EVENT_INTRUS7_OPEN_CIRCUIT	“intrusion line#7 o/c”
ID_EVENT_INTRUS6_CLOSED_CIRCUIT	“intrusion line#6 c/c”
ID_EVENT_INTRUS7_CLOSED_CIRCUIT	“intrusion line#7 c/c”
ID_EVENT OLDTIME	“current date & time”
ID_EVENT_NEWTIME	“new date & time”
ID_EVENT_INTERNALRESET_WDT	“IDLP internal reset: WDT”
ID_EVENT_INTERNALRESET_MCLR	“IDLP internal reset: MCLR”
ID_EVENT_IDLPRESETNOEE	“IDLP internal reset: no log EEPROM”
ID_EVENT_BATT1TOOLOW	“Battery 1 too low”
ID_EVENT_BATT1CORRECT	“Battery 1 correct”
ID_EVENT_BATT2TOOLOW	“Battery 2 too low”
ID_EVENT_BATT2CORRECT	“Battery 2 correct”
ID_EVENT_BATT3TOOLOW	“Battery 3 too low”
ID_EVENT_BATT3CORRECT	“Battery 3 correct”
ID_EVENT_INTERNALRESET_BOR	“IDLP internal reset: BOR”

If the event code is not recognized then the string returned is “*unknown event code #%*d.”, with the unknown event code substituted for %d.

## Notes

The strings for the return data were changed in v1.3.0: c/c was changed to closed and o/c was changed to open.

## Availability

The dpci\_id\_eventname() function first appeared in the DirectPCI API version 1.2.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
void print_event(struct idevent *inevent)
{
    const char *event_name;

    event_name = dpci_id_event_name(inevent->eventcode);
    printf("Event code: %d\n", inevent->eventcode);
    printf("Event name: %s\n", event_name);
    printf("Event time: %04d/%02d/%02d %02d:%02d:%02d\n",
           inevent->year + 2000);
           inevent->month);
           inevent->day);
           inevent->hour);
           inevent->min);
           inevent->sec);
}
```

A further example can be found in the *id\_demo.c* program.

## 2.8.11 SET IDLP DATE AND TIME

### Definition

```
#include <dpci_core_api.h>

int dpci_id_setdate(struct idevent *pdatetime);
```

### Parameters

pdatetime An *idevent* structure pointer populated with the date and time to set the real time clock to.

### Description

This function sets the date and time in the intrusion and logging processor's real-time clock in order to log events with the correct details.

The data type `struct idevent` is detailed in Figure 38: Members of struct `idevent` on p132. The `eventcode` member is ignored.

### Return Value

- |    |   |
|----|---|
| 0  | The date was set correctly.   |
| -1 | The date could not be set correctly. The DPCI core driver presently validates the fields of the supplied time and date and flags an error if any of the fields is outside of its permitted range. |

See 2.8.1 for an explanation of possible error codes.

### Notes

Prior to IDLP Firmware version 25, logging cannot occur until the date and time have been set once.

### Availability

The `dpci_id_setdate()` function first appeared in the DirectPCI API version 1.0.0.

Firmware support exists from v1.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
int myyear = 2005;
struct idevent mydate;

mydate.year = myyear - 2000;
mydate.month = 9;           /* September */
mydate.day = 20;            /* 20th */
mydate.hour = 3;
mydate.min = 15;
mydate.sec = 59;

if (dpci_id_setdate(&mydate) == -1)
{
    printf("Failure setting the real-time clock\n");
}
```

A further example can be found in the *id\_demo.c* program.

## 2.8.12 GET IDLP DATE AND TIME

### Definition

```
#include <dpci_core_api.h>

int dpci_id_getdate(struct idevent *pdatetime);
```

### Parameters

pdatetime	pointer to an <code>idevent</code> structure, as defined in the API, that will be populated with the current date and time
-----------	--

### Description

`Dpci_id_getdate()` returns the current date and time in `pdatetime`, as maintained by the intrusion detection and logging processor. The member `eventcode` will be undefined.

Details of type `struct idevent` are in Figure 38: Members of struct `idevent` on p132.

### Return Value

0	The structure passed in was populated successfully with the RTC date and time.
-1	The structure could not be populated with the date and time of the RTC. See 2.8.1 for an explanation of possible error codes.

### Availability

The `dpci_id_getdate()` function first appeared in the DirectPCI API version 1.0.0.

Firmware support exists from v1.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

struct idevent mydate;

if (dpci_id_getdate(&mydate) == -1)
{
    printf("Error getting RTC Date and Time");
}
printf("The date and time are %04d/%02d/%02d "
       "%02d:%02d:%02d\n",
       mydate.year + 2000,
       mydate.month,
       mydate.day,
       mydate.hour,
       mydate.min,
       mydate.sec);
```

A further example can be found in the `id_demo.c` program.

## 2.8.13 FIND THE LAST TIME AN EVENT OCCURRED

### Definition

```
#include <dpci_core_api.h>

int dpci_id_readevent_instance(int eventcode, struct idevent *pevent);
```

### Parameters

eventcode	The event code for which the last event is required.
pevent	pointer to an idevent structure, as defined in the API, that will be populated with event requested.

### Description

*Dpci\_id\_readevent\_instance()* returns details of the last time an event occurred.

IDLP firmware starting at v60 maintains a log of the last occurrence of many event types which is separate from the regular event log as accessed using *dpci\_id\_readevent* (see 2.8.7) and *dpci\_id\_selectevent* (see 2.8.82.8.7).

Note that:

- There is not sufficient space in the IDLP's internal RAM to record the last instance of all defined event types. Only a limited number of RAM 'slots' are available.
- The IDLP may use a given 'slot' to store more than one event type, so that on many DPX-series boards, it will not be possible to obtain the time and date of the last occurrence of every retained event code.
- The log of last events is retained in RAM. Should the IDLP's battery be replaced while the board is powered down then the log will be cleared.

The data type `struct idevent` is detailed in Figure 38: "Members of struct idevent" on p132.

### Return Value

0	Event data was returned by the IDLP successfully. You should check that the event code in the structure matches the eventcode parameter passed. If the event code does not match the parameter passed then the result is an indication that the IDLP may retain the request event code but that the space is shared with other event codes.
-1	An event of the type indicated could not be obtained. The system error code must be checked to establish the reason for this. The system error code will be set to EINVAL (Linux) or ERROR_INVALID_PARAMETER (Win32 API) if the IDLP does not support retaining the event type requested.

## Availability

The `dpci_id_readevent_instance()` function first appeared in the DirectPCI API version 3.1.0. This command requires corresponding support in the DirectPCI IDLP firmware, which is available in v60.

## See Also

The *idutil* command-line utility with option ‘t’ (4.5 ).

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>
#include <dpci_core_hw.h>      // for MAX_ID_EVENT

...
struct idevent lastevent;
int eventcode;

for (eventcode = 1; eventcode <= MAX_ID_EVENT; eventcode++)
{
    if (dpci_id_readevent_instance(eventcode, &lastevent) == -1)
    {
#ifndef WIN32
        if (dpci_last_os_error_code() == ERROR_INVALID_PARAMETER)
#else
        if (dpci_last_os_error_code() == EINVAL)
#endif
        {
            printf("Event code %d not retained in IDLP.\n", eventcode);
            continue;
        }
        perror("Error getting RTC Date and Time");
        break;
    }
    if (lastevent.eventcode != eventcode)
    {
        printf("Event code %d retained in IDLP but not available.\n",
               eventcode);
        continue;
    }
    printf("Event code %d last occurred %04d/%02d/%02d "
           "%02d:%02d:%02d\n",
           lastevent.year + 2000,
           lastevent.month,
           lastevent.day,
           lastevent.hour,
           lastevent.min,
           lastevent.sec);
}
}
```

A further example can be found in the *id\_demo.c* program.

## 2.8.14 OBTAIN THE IDLP FIRMWARE CHECKSUM

### Definition

```
#include <dpci_core_api.h>

int dpci_id_readchecksum(void);
```

### Parameters

None

### Description

*Dpci\_id\_readchecksum()* instructs the IDLP firmware to return the internal checksum calculation that is executed at firmware initialization and then retained in the IDLP's own RAM.

### Return Value

- |     |   |
|-----|---|
| >=0 | The actual checksum from the IDLP of its own firmware.  |
| -1  | An error occurred obtaining the checksum. See 2.8.1 for an explanation of possible error codes. |

### Availability

The *dpci\_id\_readchecksum()* function first appeared in the DirectPCI API version 3.1.0. This command requires corresponding support in the DirectPCI IDLP firmware, which is available in v60.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
int checksum;
checksum = dpci_id_readchecksum();
if (checksum == -1)
{
    printf("Error getting firmware checksum\n");
}
else
{
    printf("Firmware checksum: 0x%04x\n", checksum);
```

A further example can be found in the *id\_demo.c* program.

## 2.9 IDPROM API

The IDPROM is a 128-byte DS2431 one-wire device connected to the gaming logic controller (FPGA) on many DPX-series motherboards. The chip has a unique 64-bit ID whose setting is laseried into ROM during manufacturing and whose uniqueness is guaranteed across the entire one-wire/iButton device range (and thus the entire DPX-series board range). The chip also has the option for its 32-byte pages to be made permanently read-only.

The IDPROM API allows read access to the device's ID and read-write access to its memory.

Note that some or all IDPROM memory may be reserved for future use by products.

### 2.9.1 GET IDPROM ID

#### Definition

```
#include <dpci_core_api.h>

int dpci_idprom_readdir(unsigned char *buf);
```

#### Parameters

buf                    8-byte buffer to hold the IDPROM ID as returned from the driver

#### Description

*Dpci\_idprom\_readdir()* returns the 64-bit unique ID of the IDPROM device. The uniqueness is guaranteed across all one-wire devices and thus across all DPX-series boards.

#### Return Value

0	The buffer passed in was populated successfully with the IDPROM ID.
-1	An error occurred while retrieving the IDPROM ID

#### Notes

The command *dpci -u* will display the 64-bit unique ID on systems which have an IDPROM fitted.

#### Availability

The *dpci\_idprom\_readdir()* function first appeared in the DirectPCI SDK version 2.0.0.

#### Example

An example can be found in the *idprom\_demo.c* program.

## 2.9.2 GET SIZE OF IDPROM

### Definition

```
#include <dpci_core_api.h>

int dpci_idprom_size(void);
```

### Parameters

None

### Description

This function returns the size of the IDPROM in bytes.

If the board does not support an IDPROM then 0 is returned. If the board does support an IDPROM then the size of the IDPROM normally fitted to that board is returned.

### Return Value

>=0	The size of the IDPROM in bytes
-1	An error occurred

### Availability

The *dpci\_idprom\_size()* function first appeared in the DirectPCI API version 1.2.1.

### Example

```
#include <string.h>
#include <errno.h>
#include <dpci_core_api.h>
...
int size = dpci_idprom_size();
if (size <= 0)
{
    fprintf(stderr, "No IDPROM present\n");
}
else
{
    printf("IDPROM is %d bytes.\n", size);
}
```

A further example can be found in the *eeprom\_demo.c* program.

## 2.9.3 READ BYTE FROM IDPROM

### Definition

```
#include <dpci_core_api.h>

int dpci_idprom_read8(unsigned int offset, unsigned char *pvalue);
```

### Parameters

offset	The offset from the beginning of the IDPROM.
pvalue	The memory location into which to copy the requested data.

### Description

This function reads an 8-bit item of data from the IDPROM device at location *offset* and places into user memory at *pvalue*. It is an error to specify an IDPROM offset for a location not supported in physical hardware.

### Return Value

0	Success
-1	An error occurred

### Availability

The *dpci\_idprom\_read()* function first appeared in the DirectPCI SDK version 2.0.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
unsigned char data;
if (dpci_idprom_read8(0, &data) == -1)
{
    printf("Unable to read byte from IDPROM.\n");
}
```

## 2.9.4 WRITE BYTE TO IDPROM

### Definition

```
#include <dpci_core_api.h>

int dpci_idprom_write8(unsigned int offset,
                      unsigned char value);
```

### Parameters

offset	The offset from the beginning of the DirectPCI IDPROM.
value	The data to copy to the IDPROM location.

### Description

*dpci\_e2\_write8* writes the 8-bit quantity *value* to the specified DirectPCI IDPROM location *offset*. It is an error to specify an IDPROM offset for a location not supported in physical hardware.

Note that the first 256 bytes of the IDPROM are reserved for manufacturing purposes and are therefore read-only.

### Return Value

0	Operation completed successfully.
-1	Failure writing to I/O location.

### Notes

This function is only available on DPX-series main-boards with the IDPROM fitted. The size of the IDPROM may vary between different models – please consult the main-board user manual for full details.

You should exercise care when writing to an IDPROM and tailor your write-access method according to need:

- If you are writing one byte or only one byte has changed, then you should strongly consider using *dpci\_idprom\_write8* for that byte only. This saves erase cycles for unchanged bytes.
- If you need to write multiple bytes, then you should use *dpci\_idprom\_write()* as this will speed up the access considerably by using the IDPROM block transfer capabilities.

IDPROM writes typically incur a delay of milliseconds before the IDPROM can next be accessed. This delay is the same whether a write is for one byte or a whole IDPROM

page. As such, it is incredibly inefficient to initialize a whole IDPROM by performing single-byte writes.

## Availability

The *dpci\_e2\_write8()* function first appeared in the DirectPCI API version 1.2.1.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
#define IDPROM_MAGIC_LOC 0x0x7f
#define IDPROM_MAGIC_VAL 0xa6

if (dpci_e2_write8(IDPROM_MAGIC_LOC, IDPROM_MAGIC_VAL) == -1)
{
    fprintf(stderr, "Unable to write data to EEPROM.\n");
    return (-1);
}
return (0);
```

A further example can be found in the *idprom\_demo.c* program.

## 2.9.5 READ BLOCK FROM IDPROM

### Definition

```
#include <dpci_core_api.h>

int dpci_idprom_read(unsigned int offset,
                      unsigned char *buffer,
                      int datalen)
```

### Parameters

offset	The offset in memory to read from.
buffer	Pointer to memory address to store the data returned by the function.
datalen	The length of the data (in bytes) to be read from the IDPROM.

### Description

Reads *datalen* consecutive bytes of data from the DirectPCI IDPROM at the offset *offset* and copies into user memory defined by *buffer*.

The semantics of addressing are consistent with Unix/Posix read/write system calls: if the inclusive memory range [offset:offset+datalen-1] should include byte locations beyond the range physically supported, then only as many bytes are transferred as actually are possible and the transfer is truncated, the return code indicating the number of bytes actually transferred. An error code is not returned.

For example, an attempt to read 8 bytes of data from the IDPROM starting 4 bytes from the highest available memory address will transfer only four bytes and return the value 4.

### Return Value

>=0	Number of bytes read
-1	An error occurred

### Availability

The *dpci\_idprom\_read()* function first appeared in the DirectPCI SDK version 2.0.0.

### Example

```
#include <dpci_core_api.h>
char buffer [32];
int bytes_got = dpci_idprom_read(0, buffer, sizeof(buffer));
if (bytes_got < sizeof(buffer))
{
```

```
    printf("Couldn't read whole buffer from IDPROM.\n");
    exit(1);
}
```

## 2.9.6 WRITE BLOCK TO IDPROM

### Definition

```
#include <dpci_core_api.h>

int dpci_idprom_write(unsigned int offset, void *data, unsigned int len);
```

### Parameters

offset	The offset from the beginning of the DirectPCI IDPROM.
data	Pointer to first byte of data in memory.
len	The number of bytes to copy.

### Description

Writes a block of data (*len* consecutive bytes) to IDPROM from main memory.

The semantics of addressing are consistent with Unix/Posix read/write system calls: if the inclusive memory range [offset:offset+dataLEN-1] should include byte locations beyond the range physically supported, then only as many bytes are transferred as actually are possible and the transfer is truncated, the return code indicating the number of bytes actually transferred. An error code is not returned.

For example, an attempt to write 12 bytes to an offset 10 bytes before the end of the device would result in only 10 bytes being copied.

### Return Value

>0	the number of bytes written.
0	No data was read – the end of the device has been reached.
-1	Failure writing to IDPROM location.

### Notes

This function is only available on DPX-series main-boards with the IDPROM fitted.

Please refer to the notes in the section on *dpci\_idprom\_write8()* for advice on writing to IDPROM.

### Availability

The *dpci\_idprom\_write()* function first appeared in the DirectPCI API version 1.2.1.

### Example

```
#include <stdio.h>
```

```
#include <dpci_core_api.h>

...
#define IDPROM_HEADER_LOC 0x74
#define IDPROM_HEADER_MAGIC 0xdeafbabA

struct eeprom_header
{
    int magic;
    int user;
    time_t timeanddate;
} my_header;

my_header.magic = IDPROM_HEADER_MAGIC;
my_header.user = getuid();
time(&my_header.time);

ret = dpci_idprom_write(IDPROM_HEADER_LOC,
                        (void *)&my_header,
                        sizeof(my_header));
switch (ret)
{
case -1:
    fprintf(stderr, "Unable to write header to EEPROM.\n");
    return (-1);
case sizeof(my_header):
    break;
default:
    fprintf(stderr,
            "Unable to write whole header to EEPROM.\n");
    return (-1);
}
return (0);
```

A further example can be found in the *idprom\_demo.c* program.

## 2.10 DIGITAL I/O API

Note: it is recommended that for new coding projects, this API's functions not be used where it is required to wait for an event to occur; instead use the new Event Streaming API (see page 97).

### 2.10.1 GET NUMBER OF DIGITAL INPUT PORTS

#### Definition

```
#include <dpci_core_api.h>

int dpci_io_numports(int irq_capable)
```

#### Parameters

irq_capable	Determines whether or not to return the number of ports that have interrupt capabilities.
-------------	---

#### Description

Returns the number of input ports on the system. If *irq\_capable* is 0 then the total number of input ports is returned; otherwise only the number of inputs with change detection and interrupt delivery is returned. Only ports which have interrupt support can be used for event streaming or with *dpci\_io\_wait\_int()*.

Input ports are numbered logically from 0 through N-1, where N is the number available on a given platform.

Consult the user manual for your DPX-series mainboard for more information about the number of ports available and their support for interrupt-based change detection.

#### Return Value

≥0	Number of input ports supported.
-1	Failure obtaining port count.

#### Availability

The *dpci\_io\_numports()* function first appeared in the DirectPCI API version 1.4.0.

#### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
if (dpci_io_numports(1) < 2)
{
    printf("Need at least 2 ports with interrupt signalling\n");
```

}

A further example can be found in the *io\_demo.c* program.

## 2.10.2 GET NUMBER OF DIGITAL OUTPUT PORTS

### Definition

```
#include <dpci_core_api.h>

int dpci_io_numports(void)
```

### Parameters

None.

### Description

This function returns the number of output ports on the system. Output ports are numbered logically from 0 through N-1, where N is the number available on a given platform.

Consult the user manual for your DPX-series mainboard for more information about the number of ports available.

### Return Value

- |    |                                   |
|----|-----------------------------------|
| ≥0 | Number of output ports supported. |
| -1 | Failure obtaining port count.     |

### Availability

The *dpci\_io\_numports()* function first appeared in the DirectPCI API version 1.4.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
if (dpci_io_numports() == 0)
{
    printf("Need at least 1 output port!\n");
}
```

A further example can be found in the *io\_demo.c* program.

### 2.10.3 READ DIGITAL INPUT PORT

#### Definition

```
#include <dpci_core_api.h>

int dpci_io_read_port(int port_no,
                      unsigned char *pvalue);
```

#### Parameters

- |         |  |
|---------|--|
| Port_no | The input port from which to read.                         |
| Pvalue  | The memory location into which to copy the requested data. |

#### Description

Reads an 8-bit quantity from the specified DirectPCI I/O input port given by *port\_no* and copies it to the user-specified location *pvalue*.

Input and output ports are numbered logically and are always 0-based; they are not the same as register numbers and are intended to allow easier cross-platform porting. Port numbers for your DPX-series board are listed in the board's User Manual.

The following table lists the relationship between port numbers and DI pin numbers:

Figure 42. Input Ports and DI Pin Numbers

Port number	DI pins
0	DI0-DI7
1	DI8-DI15
2	DI16-DI23
3	DI24-DI31

#### Return Value

- |    |                                   |
|----|-----------------------------------|
| 0  | Operation completed successfully. |
| -1 | Failure writing to I/O location.  |

#### Notes

It is strongly suggested this function be used instead of the *dpci\_io\_readN()* functions so that inadvertent accesses to critical system registers are avoided.

## Availability

The *dpci\_io\_read\_port()* function first appeared in the DirectPCI API version 1.3.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
#define INPUT1 0x1
unsigned char data = 0;
if (dpci_io_read_port(DINPUT1, &data) == -1)
{
    printf("Unable to read data from input port %d.\n", INPUT1);
```

A further example can be found in the *io\_demo.c* program.

## 2.10.4 WRITE DIGITAL OUTPUT PORT

### Definition

```
#include <dpci_core_api.h>

int dpci_io_write_port(int port_no, unsigned char value);
```

### Parameters

<b>port_no</b>	The output port number to which to write data.
<b>value</b>	The data to write to the output port.

### Description

Writes the 8-bit quantity *value* to the output port *port\_no*.

Input and output ports are numbered logically and are always 0-based; they are not the same as register numbers and are intended to allow easier cross-platform porting. Port numbers for your DPX-series board are listed in the board's User Manual.

The following table lists the relationship between port numbers and DO pin numbers:

Figure 43. Output Ports and DO Pin Numbers

<b>Port number</b>	<b>DO pins</b>
0	DO0-DO7
1	DO8-DO15
2	DO16-DO23
3	DO24-DO31
4	DO32-DO39 (on 80-0062 I/O Board II only)

### Return Value

0	Operation completed successfully.
-1	Failure writing to output port.

### Notes

It is strongly suggest this function be used instead of *dpci\_io\_write\**() functions so that inadvertent accesses to critical system registers are avoided.

## Availability

The *dpci\_io\_write\_port()* function first appeared in the DirectPCI API version 1.3.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

#define MY_OP_PORT 3

...
if (dpci_io_write_port(MY_OP_PORT, 0xFF) == -1)
{
    printf("Unable to write byte to output port.\n");
}
```

A further example can be found in the *io\_demo.c* program.

## 2.10.5 CHANGE DIGITAL OUTPUT PORT

### Definition

```
#include <dpci_core_api.h>

int dpci_io_change_port(int port_no,
                        unsigned char set,
                        unsigned char clear,
                        unsigned char toggle);
```

### Parameters

port_no	The output port number whose bit pattern is to be changed.
set	The bit mask of the bits to be set
clear	The bit mask of the bits to be cleared
toggle	The bit mask of the bits to be toggled

### Description

This function changes an output port determined by `port_no` by setting the bits specified by `set`, clearing the bits specified by `clear` and then inverting the bits specified by `toggle`.

The operation is equivalent to the following code:

```
change_port(int portno,
            unsigned char set,
            unsigned char clear,
            unsigned char toggle)
{
    unsigned char data;

    dpci_io_read_outport(portno, &data)
    data &= ~clear;
    data |= set;
    data ^= toggle;
    dpci_io_write_port(portno, data);
}
```

The three logical operations are implemented in the order shown to produce one single write access to the output register for the port.

Input and output ports are numbered logically and are always 0-based; they are not the same as register numbers and are intended to allow easier cross-platform porting. Port numbers for your DPX-series board are listed in the board's User Manual.

### Return Value

0	Operation completed successfully.
---	-----------------------------------

-1 Failure setting output port bits

## **Availability**

The *dpci\_io\_change\_port()* function first appeared in the DirectPCI API version 2.0.0.

## **Example**

An example can be found in the *io\_demo.c* program.

## 2.10.6 READ DIGITAL OUTPUT PORT

### Definition

```
#include <dpci_core_api.h>

int dpci_io_read_outport(int port_no, unsigned char *value);
```

### Parameters

port_no	The output port number to read from.
value	Buffer to hold the data read from the output port.

### Description

Reads the last 8-bit value written to the output port *port\_no*.

Input and output ports are numbered logically and are always 0-based; they are not the same as register numbers and are intended to allow easier cross-platform porting. Port numbers for your DPX-series board are listed in the board's User Manual.

### Return Value

0	Operation completed successfully.
-1	Failure retrieving last byte written to output port.

### Notes

This function does not read from the output port. The output ports are write-only and this function is provided to retrieve the last byte written to the output port. The data is retrieved from the data-structure used to hold copies of write-only registers in the driver.

### Availability

The *dpci\_io\_read\_outport()* function first appeared in the DirectPCI API version 2.0.0.

### Example

An example can be found in the *io\_demo.c* program.

## 2.10.7 WAIT FOR AN EDGE-TRIGGERED INTERRUPT ON A DIGITAL INPUT PORT

### Definitions

```
#include <dpci_core_api.h>

int dpci_io_wait_port(int port_no,
                      unsigned char int_mask,
                      unsigned char edge_state,
                      unsigned long timeout_ms);
```

### Parameters

port_no	Select which port to wait on.
int_mask	Mask specifying which lines on the given port to wait for.
edge_state	Mask specifying the type of edges to wait for.
timeout_ms	Maximum time in milliseconds (ms) to wait for an interrupt.

### Description

This function configures the given port's interrupt detection registers and waits for an interrupt to occur. The function returns when either an interrupt has occurred for at least one of the requested edges or when the timeout has elapsed.

The *int\_mask* parameter is a bit-mask of those input lines for which interrupt detection should be performed.

For each bit that is set in *int\_mask*, the corresponding bit in *edge\_state* determines the type of edge that will be detected: a 1 in that position in *edge\_state* causes a rising edge to be detected, a 0 causes a falling edge to be detected.

The timeout parameter *timeout\_ms* is counted in milliseconds. A setting of 0 causes the function to return almost immediately. A setting of -1 means  $2^{32}-1$  ms – an effective timeout of 4.29 million seconds – 49.7 days. While this might be considered to be near-infinite for some applications, applications should code for the possibility that a time-out condition might still occur.

### Return Value

>0	Bit-mask of bits from <i>int_mask</i> which have changed.
0	Operation timed-out waiting for an interrupt.
-1	Failure waiting for the interrupt.

## Notes

Although it is possible for more than one process to use this function concurrently, it is not possible for two processes to wait for interrupts on the same input line with the same or different edge settings.

Calling with *int\_mask* set to zero causes *dpci\_io\_wait\_port()* to return 0 once the specified time-out has elapsed. Due to host OS latencies, it is not recommended that this function be used to implement reliable delay mechanisms.

The accuracy of the time-out is limited by the scheduling capabilities of the host operating system. As such the actual delay may be more or less than the specified time-out value.

It is recommended that new code be developed using the event-streaming API (p97) in preference to this function.

It is recommended that users not attempt to use both this facility and the event-streaming API facility simultaneously, as the two facilities will unsuccessfully attempt to share the same interrupt programming registers.

## Availability

The *dpci\_io\_wait\_port()* function first appeared in the DirectPCI API version 1.3.0.

## Example

This code examples demonstrates how to use *dpci\_io\_wait\_port* and *dpci\_io\_read\_port* to report changes to an input port's input lines.

```
#include <stdio.h>
#include <dpci_core_api.h>
```

```
...
#define MY_PORT          1
#define INPUT_BUTTONS_MASK 0x1c
#define INPUT_TIMEOUT_MS   30000 /* 30 seconds */

while (1)
{
    /*
     * Start off expecting rising edges.
     */
    static unsigned char edge_state = INPUT_BUTTONS_MASK;
    int int_mask;
    unsigned char data;

    int_mask = dpci_io_wait_port(MY_PORT,
                                INPUT_BUTTONS_MASK,
                                edge_state,
                                INPUT_TIMEOUT_MS);

    if (int_mask < 0)
    {
        perror("dpci_io_wait_int()");
        break;
    }
    if (int_mask == 0)
    {
        printf("Timed-out!\n");
        continue;
    }

    dpci_io_read_port(MY_PORT, &data);
    printf("change-mask=%02x data=%02x\n", int_mask, data);

    /*
     * Toggle the edge state bits so that the next (opposite)
     * edges can now be detected.
     */
    edge_state ^= int_mask;
}
```

## 2.10.8 WAIT FOR AN EDGE-TRIGGERED INTERRUPT ON DIGITAL INPUT PORT #0 (DIN0-7)

### Definitions

```
#include <dpci_core_api.h>

int dpci_io_wait_int(unsigned char int_mask,
                     unsigned char edge_state,
                     unsigned long timeout_ms);
```

### Parameters

int_mask	Select which bits of DIN0-DIN7 can generate an interrupt.
edge_state	Mask specifying the type of edge to wait for.
timeout_ms	Maximum time in milliseconds (ms) to wait for an interrupt.

### Description

This function configures input port 0's interrupt detection registers and waits for an interrupt to occur. In every other respect it is identical to *dpci\_io\_wait\_port()* which performs this function for any available input port.

### Return Value

>0	Bit-mask of bits from <i>int_mask</i> which have changed.
0	Operation timed-out waiting for an interrupt.
-1	Failure waiting for the interrupt.

### Availability

The *dpci\_io\_wait\_int()* function first appeared in the DirectPCI API version 1.2.1.

### Example

See the example for *dpci\_io\_wait\_port()*.

## 2.10.9 WAIT FOR AN INTERRUPT ON DIGITAL INPUT PORT

### Definitions

```
#include <dpci_core_api.h>

int dpci_io_wait_iport(int iPort,
                      unsigned char byMask,
                      unsigned char byEdgeState,
                      unsigned char byAutoconfigEdge,
                      unsigned long dwTimeout,
                      struct dio_port_event *digip_data);
```

### Parameters

iPort	Select which port to wait on
byMask	Mask specifying which lines on the given port to wait for.
byEdgeState	Mask specifying the type of edge to wait for.
byAutoconfigEdge	Mask specifying which digital inputs should automatically be reconfigured to wait for interrupts for the opposite edge type once an interrupt has occurred
dwTimeout	Maximum time in milliseconds (ms) to wait for an interrupt.
digip_data	Buffer to hold the event signaled. The <i>dio_port_event</i> structure is defined in <i>dpci_core_types.h</i>

### Description

This function configures the given port's interrupt detection registers and waits for an interrupt to occur. When an interrupt occurs it will return the event data in the *dio\_port\_event* buffer and will automatically reconfigure the inputs set in the *byAutoconfigEdge*. In every other respect it is identical to *dpci\_io\_wait\_port()* which performs this function for any available input port.

### Return Value

1	Success. Interrupt was detected.
0	Operation timed-out waiting for an interrupt.
-1	Failure waiting for the interrupt.

### Availability

The *dpci\_io\_wait\_iport()* function first appeared in the DirectPCI API version 3.0.0.

## **Example**

An example can be found in the *io\_demo.c* program.

## 2.10.10 WAIT FOR AN INTERRUPT ON ALL DIGITAL INPUT PORTS

### Definitions

```
#include <dpci_core_api.h>

int dpci_io_wait_iports(unsigned long Mask,
                       unsigned long EdgeState,
                       unsigned long AutoconfigEdge,
                       unsigned long dwTimeout,
                       struct dio_event *dio_data);
```

### Parameters

Mask	Bit mask specifying which lines on the ports to wait for.
EdgeState	Bit mask specifying the type of edge to wait for.
AutoconfigEdge	Bit mask specifying which digital inputs should automatically be reconfigured to wait for interrupts for the opposite edge type once an interrupt has occurred
dwTimeout	Maximum time in milliseconds (ms) to wait for an interrupt.
dio_data	Buffer to hold the event signaled. The dio_event structure is defined in dpci_core_types.h

### Description

This function configures the all digital input ports' interrupt detection registers and waits for an interrupt to occur. When an interrupt occurs it will return the event data in the *dio\_event* buffer and will automatically reconfigure the inputs set in the *AutoconfigEdge*. In every other respect it is identical to *dpci\_io\_wait\_iport()* which performs this function for a selected input port. Bits 0-7 correspond to input port 0, bits 8-15 to input port 1 and etc.

### Return Value

1	Success. Interrupt was detected.
0	Operation timed-out waiting for an interrupt.
-1	Failure waiting for the interrupt.

### Availability

The *dpci\_io\_wait\_iports()* function first appeared in the DirectPCI API version 3.0.0.

## Example

See the example for *dpci\_io\_wait\_port()*.

## 2.10.11 EDGE-TRIGGERED DIGITAL INPUT EVENTS (LINUX ONLY, DEPRECATED)

### Definition

```
#include <fcntl.h>
#include <linux/input.h>

int fd;
struct input_event ev;

fd = open("/dev/input/dpci", O_RDONLY);
read(fd, (char *)&ev, sizeof(ev));
```

### Description

The Linux 2.6 kernel series contains a mechanism for reporting events on keyboards, mice, joysticks and other input devices. This event based mechanism can also be used to obtain time-stamped information about interrupts occurring on digital input ports.

Events can be obtained by first opening the appropriate input device (see notes and example below) and then using the *read(2)* system call. The read system call populates user-supplied instances of *struct input\_event* which is reproduced here.

```
struct input_event
{
    struct timeval time;
    __u16 type;
    __u16 code;
    __s32 value;
}
```

Figure 44. Linux Native Input Device – *struct input\_event*

The *time* field reports the Linux kernel time at which the event was reported to the kernel – this is the time at which the interrupt handler was invoked, which may be delayed from the time of the actually input change.

The *type* field reports the type of the event – ordinarily this will be always EV\_KEY.

The *code* field reports a code for the input that caused the event. In the current implementation the following mapping is used.

Input	Event code	Event code Value
DIN0	BTN_0	256
DIN1	BTN_1	257
DIN2	BTN_2	258
DIN3	BTN_3	259
DIN4	BTN_4	260

Input	Event code	Event code Value
DIN5	BTN_5	261
DIN6	BTN_6	262
DIN7	BTN_7	263

Figure 45. Event Codes numbers for DirectPCI Digital Inputs

The *value* field contains the newly latched value read from the input port 0 register.

## Notes

It is recommended this feature not be used for new code; instead use the Event Streaming API described from page 97.

This interface is only supported on Linux kernels in the 2.6.x series. The kernel must be configured with the following parameters defined:

`CONFIG_INPUT_EVDEV=m`

or

`CONFIG_INPUT_EVDEV=y`

Ordinarily the input device is named `/dev/input/eventn` where *n* is dependent upon the input modules already loaded when the DirectPCI kernel module (`dpci_core.ko`) is loaded.

To find the correct device to open, one can either configure the *udev* facility so that `/dev/input/dpci` is created automatically or one can parse the contents of `/proc/bus/input/devices` to find the entry like this.

```
I: Bus=0001: Vendor=16cd Product=0105 Version=0003
N: Name="Innocore DirectPCI Input0"
P: Phys=dpci/input0
H: Handler=event1
B: EV=3
B: KEY=ff 0 0 0 0 0 0 0 0 0
```

To configure *udev* one can simply add a single line like this to `/etc/udev/rules.d/udev.rules`. (Note that the line should be entered as a single line without the continuation character \ shown below.)

```
KERNEL="event*" SYSFS{vendor}="0x16cd", NAME="input/%k", \
SYMLINK="input/dpci"
```

## Availability

This facility first appeared in the DirectPCI API version 1.1.0.

This facility is now deprecated and will be removed in the next minor (3.x) release of this product in favour of the new event-streaming API.

## Example

```
#include <stdio.h>
#include <fcntl.h>
#include <linux/input.h>

int fd;
struct input_event ev;

fd = open("/dev/input/dpci", O_RDONLY);
if (fd == -1)
{
    ...
}
if (dpci_io_config_input(0, 0x8, 0x8) == -1)
{
    ...
}
while (1)
{
    int got;

    got = read(fd, (void *)&ev, sizeof(ev));
    if (got != sizeof(ev))
    {
        perror("read()");
        break;
    }
    printf("Din%d is %d\n", ev.code - BTN_0, ev.value);
    ...
}
```

A further example can be found in the *io\_demo.c* program.

## 2.10.12 READ BYTE FROM DIRECTPCI REGISTER (DEPRECATED)

### Definition

```
#include <dpci_core_api.h>

int dpci_io_read8(dpci_off_t offset,
                  unsigned char *pvalue);
```

### Parameters

offset	The offset from the base address of the DirectPCI device.
Pvalue	The memory location into which to copy the requested data.

### Description

Reads an 8-bit quantity from the specified DirectPCI I/O location *offset* and copies it to the user-specified location *pvalue*.

### Return Value

0	Operation completed successfully.
-1	Failure writing to I/O location.

### Availability

The *dpci\_io\_read8()* function first appeared in the DirectPCI API version 1.2.0.

This function is now deprecated and may be removed in a future version.

You must set debug level 0x8000 to use this function.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
#define DINPUT1 0x16
unsigned char data = 0;
if (dpci_io_read8(DINPUT1, &data) == -1)
{
    printf("Unable to read data from I/O location.\n");
```

A further example can be found in the *io\_demo.c* program.

## 2.10.13 READ WORD FROM DIRECTPCI REGISTER (DEPRECATED)

### Definition

```
#include <dpci_core_api.h>

int dpci_io_read16(dpci_off_t offset,
                    unsigned short *pvalue);
```

### Parameters

offset	The offset from the base address of the DirectPCI device.
Pvalue	The memory location into which to copy the requested data

### Description

Reads a 16-bit quantity from the specified DirectPCI I/O location *offset* and copies it to the user-specified location *pvalue*.

### Return Value

0	Operation completed successfully.
-1	Failure writing to I/O location.

### Availability

The *dpci\_io\_read16()* function first appeared in the DirectPCI API version 1.2.0.

This function is now deprecated and may be removed in a future version.

You must set debug level 0x8000 to use this function.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
#define DINPUT1 0x16
unsigned short data = 0;
if (dpci_io_read16(DINPUT1, &data) == -1)
{
    printf("Unable to read data from I/O location.\n");
```

A further example can be found in the *io\_demo.c* program.

## 2.10.14 READ DOUBLE WORD FROM DIRECTPCI REGISTER (DEPRECATED)

### Definition

```
#include <dpci_core_api.h>

int dpci_io_read32(dpci_off_t offset,
                    unsigned int *pvalue);
```

### Parameters

offset	The offset from the base address of the DirectPCI device.
Pvalue	The memory location into which to copy the requested data

### Description

Reads a 32-bit quantity from the specified DirectPCI I/O location *offset* and copies it to the user-specified location *pvalue*.

### Return Value

-1	An error occurred access the I/O location.
0	The I/O location was accessed successfully.

### Availability

The *dpci\_io\_read32()* function first appeared in the DirectPCI API version 1.2.0.

This function is now deprecated and may be removed in a future version.

You must set debug level 0x8000 to use this function.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
unsigned int data = 0;
if (dpci_io_read32(DPCI_IOBOARD_IPO, &data) == -1)
{
    printf("unable to read data from I/O location.\n");
}
```

A further example can be found in the *io\_demo.c* program.

## 2.10.15 WRITE BYTE TO DIRECTPCI REGISTER (DEPRECATED)

### Definition

```
#include <dpci_core_api.h>

int dpci_io_write8(dpci_off_t offset, unsigned char value);
```

### Parameters

**offset** The offset from the base address of the DirectPCI device.

**value** The value to write to the I/O location.

### Description

Writes the 8-bit quantity given by *value* to the I/O location *offset* from the DirectPCI base address.

### Return Value

0 Operation completed successfully.

-1 Failure writing to I/O location.

### Notes

Exercise caution when writing to registers: incorrect accesses can cause system failures. In particular, avoid accessing registers directly which are concerned with interrupt management.

### Availability

The *dpci\_io\_write8()* function first appeared in the DirectPCI API version 1.2.0.

This function is now deprecated and may be removed in a future version.

You must set debug level 0x8000 to use this function.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

... if (dpci_io_write8(DPCI_IOBOARD_OPO, 0xFF) == -1)
    printf("Unable to write byte to I/O location.\n");
```

A further example can be found in the *io\_demo.c* program.

## 2.10.16 WRITE WORD TO DIRECTPCI REGISTER (DEPRECATED)

### Definition

```
#include <dpci_core_api.h>

int dpci_io_write16(dpci_off_t offset, unsigned short value);
```

### Parameters

offset	The offset from the base address of the DirectPCI device.
value	The value to write to the I/O location.

### Description

Writes the 16-bit quantity given by *value* to the I/O location at *offset* from the DirectPCI base address.

### Return Value

0	Operation completed successfully.
-1	Failure writing to I/O location.

### Notes

Exercise caution when writing to registers: incorrect accesses can cause system failures. In particular, avoid accessing registers directly which are concerned with interrupt management.

### Availability

The *dpci\_io\_write16()* function first appeared in the DirectPCI API version 1.2.0.

This function is now deprecated and may be removed in a future version.

You must set debug level 0x8000 to use this function.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
if (dpci_io_write16(DPCI_IOBOARD_OP2, 0xbeef) == -1)
{
    printf("Unable to write word to I/O location.\n");
}
```

A further example can be found in the *io\_demo.c* program.

## 2.10.17 WRITE DOUBLE WORD TO DIRECTPCI REGISTER (DEPRECATED)

### Definition

```
#include <dpci_core_api.h>

int dpci_io_write32(dpci_off_t offset, unsigned int value);
```

### Parameters

offset	The offset from the base address of the DirectPCI device.
value	The value to write to the I/O location.

### Description

Writes the 32-bit quantity given by *value* to the I/O location at *offset* from the DirectPCI base address.

### Return Value

0	Operation completed successfully.
-1	Failure writing to I/O location.

### Notes

Exercise caution when writing to registers: incorrect accesses can cause system failures. In particular, avoid accessing registers directly which are concerned with interrupt management.

### Availability

The `dpci_io_write32()` function first appeared in the DirectPCI API version 1.2.0.

This function is now deprecated and may be removed in a future version.

You must set debug level 0x8000 to use this function.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
if (dpci_io_write32(DPCI_IOBOARD_OP0, 0xdeadbeef) == -1)
{
    printf("Unable to write dword to I/O location.\n");
}
```

A further example can be found in the `io_demo.c` program.

## 2.11 MOTHERBOARD EEPROM API

### 2.11.1 GET SIZE OF EEPROM

#### Definition

```
#include <dpci_core_api.h>

int dpci_e2_size(void);
```

#### Parameters

None

#### Description

This function returns the size of the motherboard EEPROM in bytes.

If the board does not support an EEPROM then 0 is returned. If the board does support an EEPROM then the size of the EEPROM normally fitted to that board is returned. It is not possible at present to determine the actual size of the EEPROM fitted without destructive testing.

#### Return Value

>=0	The size of the EEPROM in bytes
-1	An error occurred

#### Availability

The *dpci\_e2\_size()* function first appeared in the DirectPCI API version 1.2.1.

#### Example

```
#include <string.h>
#include <errno.h>
#include <dpci_core_api.h>
...
int size = dpci_e2_size();
if (size <= 0)
{
    fprintf(stderr, "No EEPROM present\n");
}
```

A further example can be found in the *eeprom\_demo.c* program.

## 2.11.2 READ BYTE FROM EEPROM

### Definition

```
#include <dpci_core_api.h>

int dpci_e2_read8(unsigned int offset,
                  unsigned char *pvalue);
```

### Parameters

offset	The offset from the beginning of the DirectPCI EEPROM.
Pvalue	The memory location into which to copy the requested data.

### Description

Reads an 8-bit quantity from the specified DirectPCI EEPROM location *offset* and copies it to a user-specified location *pvalue*. It is an error to specify an EEPROM offset for a location not supported in physical hardware.

### Return Value

0	Operation completed successfully.
-1	Failure writing to I/O location.

### Notes

This function is only available on DPX-series main-boards with the EEPROM fitted. The size of the EEPROM may vary between different models – please consult the main-board User Manual for full details.

When reading data from adjacent locations in the EEPROM, it is considerably more efficient to use *dpci\_e2\_read()*.

This function can only be used to access the main-board EEPROM. The function *dpci\_i2c\_read()* should be used to access other I<sup>2</sup>C devices.

In the event of an error, the function *dpci\_i2c\_last\_error()* (p206) can be used to obtain the I<sup>2</sup>C error code from the operation.

In version 1.3.0 of the DirectPCI API, all EEPROM access functions are layered on top of the I<sup>2</sup>C access library functions. Previous Linux versions implemented this functionality in the *dpci\_core* device driver.

### Availability

The *dpci\_e2\_read8()* function first appeared in the DirectPCI API version 1.2.1.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
#define EEPROM_MAGIC_LOC 0x100
#define EEPROM_MAGIC_VAL 0xa6

unsigned char data = 0;

if (dpci_e2_read8(EEPROM_MAGIC_LOC, &data) == -1)
{
    fprintf(stderr, "Unable to read data from I/O location.\n");
    return (-1);
}
if (data != EEPROM_MAGIC_VAL)
{
    fprintf(stderr, "MAGIC number in EEPROM is wrong.\n");
    return (-1);
}
return (0);
```

A further example can be found in the *eeprom\_demo.c* program.

## 2.11.3 WRITE BYTE TO EEPROM

### Definition

```
#include <dpci_core_api.h>

int dpci_e2_write8(unsigned int offset,
                    unsigned char value);
```

### Parameters

offset	The offset from the beginning of the DirectPCI EEPROM.
value	The data to copy to the EEPROM location.

### Description

*dpci\_e2\_write8* writes the 8-bit quantity *value* to the specified DirectPCI EEPROM location *offset*. It is an error to specify an EEPROM offset for a location not supported in physical hardware.

### Return Value

0	Operation completed successfully.
-1	Failure writing to I/O location.

### Notes

This function is only available on DPX-series main-boards with the EEPROM fitted. The size of the EEPROM may vary between different models – please consult the main-board user manual for full details.

You should exercise care when writing to an EEPROM and tailor your write-access method according to need:

- If you are writing one byte or only one byte has changed, then you should strongly consider using *dpci\_e2\_write8* for that byte only. This saves erase cycles for unchanged bytes.
- If you need to write multiple bytes, then you should use *dpci\_e2\_write()* as this will speed up the access considerably for most EEPROM types by using the EEPROM's page write capabilities.

EEPROMs writes typically incur a delay of 10s of milliseconds before the EEPROM can next be accessed. This delay is the same whether a write is for one byte or a whole EEPROM page. As such, it is incredibly inefficient to initialize a whole EEPROM by performing single-byte writes.

This function can only be used to access the main-board EEPROM. The function *dpci\_i2c\_write()* should be used to access other I<sup>2</sup>C devices.

In the event of an error, the function `dpci_i2c_last_error()` (p206) can be used to obtain the I<sup>2</sup>C error code from the operation.

In version 1.3.0 of the DirectPCI API, all EEPROM access functions are layered on top of the I<sup>2</sup>C access library functions. Previous Linux versions implemented this functionality in the `dpci_core` device driver.

## Availability

The `dpci_e2_write8()` function first appeared in the DirectPCI API version 1.2.1.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
#define EEPROM_MAGIC_LOC 0x100
#define EEPROM_MAGIC_VAL 0xa6

if (dpci_e2_write8(EEPROM_MAGIC_LOC, EEPROM_MAGIC_VAL) == -1)
{
    fprintf(stderr, "Unable to write data to EEPROM.\n");
    return (-1);
}
return (0);
```

A further example can be found in the `eeprom_demo.c` program.

## 2.11.4 READ BLOCK FROM EEPROM

### Definition

```
#include <dpci_core_api.h>

int dpci_e2_read(unsigned int offset, void *data, unsigned int len);
```

### Parameters

offset	The offset from the beginning of the DirectPCI EEPROM.
data	Pointer to memory where the first byte of data will be copied.
len	The number of bytes to copy.

### Description

Reads a block of data (*len* consecutive bytes) from EEPROM into main memory locations chosen by the user.

The semantics of addressing are consistent with Unix/Posix read/write system calls: if the inclusive memory range [offset:offset+dataLEN-1] should include byte locations beyond the range physically supported, then only as many bytes are transferred as actually are possible and the transfer is truncated, the return code indicating the number of bytes actually transferred. An error code is not returned.

For example, an attempt to read 12 bytes from an offset 10 bytes before the end of the device would result in only 10 bytes being returned.

### Return Value

>0	the number of bytes read.
0	No data was read – the end of the device has been reached.
-1	Failure writing to I/O location.

### Notes

This function is only available on DPX-series main-boards with the EEPROM fitted. The size of the EEPROM may vary between different models – please consult the main-board User Manual for full details.

This function can only be used to access the main-board EEPROM. The function *dpci\_i2c\_read()* should be used to access other EEPROMs and I<sup>2</sup>C devices.

In the event of an error, the function *dpci\_i2c\_last\_error()* (p206) can be used to obtain the I<sup>2</sup>C error code from the operation.

In version 1.3.0 of the DirectPCI API, all EEPROM access functions are layered on top of the I<sup>2</sup>C access library functions. Previous Linux versions implemented this functionality in the *dpci\_core* device driver.

## Availability

The *dpci\_e2\_read()* function first appeared in the DirectPCI API version 1.2.1.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
#define EEPROM_HEADER_LOC 0x100
struct eeprom_header my_header;
ret = dpci_e2_read(EEPROM_HEADER_LOC,
                   (void *)&my_header,
                   sizeof(my_header));
switch (ret)
{
case -1:
    fprintf(stderr, "Unable to read header from EEPROM.\n");
    return (-1);
case sizeof(my_header):
    break;
default:
    fprintf(stderr,
            "Unable to read whole header from EEPROM.\n");
    return (-1);
}
return (0);
```

A further example can be found in the *eeprom\_demo.c* program.

## 2.11.5 WRITE BLOCK TO EEPROM

### Definition

```
#include <dpci_core_api.h>

int dpci_e2_write(unsigned int offset, void *data, unsigned int len);
```

### Parameters

offset	The offset from the beginning of the DirectPCI EEPROM.
data	Pointer to first byte of data in memory.
len	The number of bytes to copy.

### Description

Writes a block of data (*len* consecutive bytes) to EEPROM from main memory.

The semantics of addressing are consistent with Unix/Posix read/write system calls: if the inclusive memory range [offset:offset+dataLEN-1] should include byte locations beyond the range physically supported, then only as many bytes are transferred as actually are possible and the transfer is truncated, the return code indicating the number of bytes actually transferred. An error code is not returned.

For example, an attempt to write 12 bytes to an offset 10 bytes before the end of the device would result in only 10 bytes being copied.

### Return Value

>0	the number of bytes written.
0	No data was read – the end of the device has been reached.
-1	Failure writing to eeprom location.

### Notes

This function is only available on DPX-series main-boards with the EEPROM fitted. The size of the EEPROM may vary between different models – please consult the main-board User Manual for full details.

This function can only be used to access the main-board EEPROM. The function *dpci\_i2c\_read()* should be used to access other I<sup>2</sup>C devices.

Please refer to the notes in the section on *dpci\_e2\_write8()* for advice on writing to EEPROMs.

In the event of an error, the function *dpci\_i2c\_last\_error()* (p206) can be used to obtain the I<sup>2</sup>C error code from the operation.

In version 1.3.0 of the DirectPCI API, all EEPROM access functions are layered on top of the I<sup>2</sup>C access library functions. Previous Linux versions implemented this functionality in the *dpci\_core* device driver.

## Availability

The *dpci\_e2\_write()* function first appeared in the DirectPCI API version 1.2.1.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...

#define EEPROM_HEADER_LOC 0x100
#define EEPROM_HEADER_MAGIC 0xdeafbaba

struct eeprom_header
{
    int magic;
    int user;
    time_t timeanddate;
} my_header;

my_header.magic = EEPROM_HEADER_MAGIC;
my_header.user = getuid();
time(&my_header.time);

ret = dpci_e2_write(EEPROM_HEADER_LOC,
                    (void *)&my_header,
                    sizeof(my_header));
switch (ret)
{
case -1:
    fprintf(stderr, "Unable to write header to EEPROM.\n");
    return (-1);
case sizeof(my_header):
    break;
default:
    fprintf(stderr,
            "Unable to write whole header to EEPROM.\n");
    return (-1);
}
return (0);
```

A further example can be found in the *eeprom\_demo.c* program.

## 2.12 I<sup>2</sup>C API

### 2.12.1 GET NUMBER OF I<sup>2</sup>C BUSES

#### Definitions

```
#include <dpci_core_api.h>

int dpci_i2c_numbuses();
```

#### Parameters

None.

#### Description

*Dpci\_i2c\_numbuses* allows the programmer to retrieve the number of I<sup>2</sup>C buses supplied by the system's DirectPCI hardware.

#### Return Value

- |     |  |
|-----|--|
| >=0 | The number of I <sup>2</sup> C buses supplied by DirectPCI hardware. |
| -1  | An error occurred while obtaining the number of buses.               |

#### Notes

See the description of *dpci\_i2c\_bus\_name()* for more information on the buses that are available on DPX-series mainboards.

#### Availability

The *dpci\_i2c\_numbuses()* function first appeared in the DirectPCI API version 1.4.0.

#### Example

See the example code for *dpci\_i2c\_bus\_name()*. Further examples can be found in the *i2c\_demo.c* program.

## 2.12.2 GET BUS NUMBER FOR A NAMED I<sup>2</sup>C BUS

### Definitions

```
#include <dpci_core_api.h>

int dpci_i2c_bus_number(char *name);
```

### Parameters

name	The name of the bus whose logical number is to be obtained.
------	---

### Description

*Dpci\_i2c\_bus\_number()* allows the programmer to retrieve the number of an I<sup>2</sup>C bus whose name is known.

### Return Value

>=0	The number of the bus whose name is in <i>name</i> .
-1	An error occurred while obtaining the bus's name.

### Notes

Note that name comparison is presently case-sensitive. See the description of *dpci\_i2c\_bus\_name()* for more information on the buses that are available on DPX-series mainboards.

### Availability

The function *dpci\_i2c\_bus\_number()* first appeared in the DirectPCI API version 1.4.0.

### Example

```
#include <dpci_core_api.h>
#include <stdio.h>

int find_bus_or_die(char *name)
{
    int bus;

    bus = dpci_i2c_bus_number(name);
    if (bus == -1)
    {
        fprintf(stderr, "Bus named %s is not available.\n", name);
        exit(1);
    }
    printf("Bus number for %s is %d.\n", name, bus);
}
```

Further examples can be found in the *i2c\_demo.c* program.

### 2.12.3 GET I<sup>2</sup>C BUS NAME

#### Definitions

```
#include <dpci_core_api.h>

int dpci_i2c_bus_name(int bus, char *buf, unsigned int bufsiz);
```

#### Parameters

- |        |  |
|--------|--|
| bus    | The bus number whose name is to be returned            |
| buf    | The buffer into which the bus name should be returned. |
| bufsiz | The size of the buffer <i>buf</i> .                    |

#### Description

*Dpci\_i2c\_bus\_name* allows the programmer to retrieve the logical name for a given I<sup>2</sup>C bus number. The name is returned in the buffer pointed to by *buf*. At most *bufsiz* bytes are copied to *buf*, including a NUL-terminator. Generally, the names assigned to I<sup>2</sup>C busses are less than 16 characters long.

#### Return Value

- |    |   |
|----|---|
| 0  | The name was returned correctly. It may have been truncated if <i>bufsiz</i> was too small. |
| -1 | An error occurred while obtaining the bus's name.   |

#### Notes

This function is only usable where I<sup>2</sup>C bus facilities are available via DirectPCI hardware. The following table shows the I<sup>2</sup>C bus support for various models of DPX-series mainboards.

Figure 46. I<sup>2</sup>C Busses on DPX-series Mainboards

Main-board	On-board busses	External busses
DPX-E105	#0: MB/EEPROM	As for 80-0062 but not used.
DPX-Cx05	#0: MB/EEPROM	With 80-0062 I/O Board

Main-board	On-board busses	External busses
DPX-S410	#0: MB/EEPROM/EXP	None
DPX-S305		
DPX-S420		
DPX-S415		
DPX-S425		

The on-board bus gives access to the motherboard EEPROM which can be accessed using the *dpci\_e2\_\**() functions. On some boards this bus can have additional hardware connected (e.g. via the expansion connector on the DPX-112 and via the backplane connector on the DPX-116U).

The 80-0062 I/O Board supplies the following additional off-board I<sup>2</sup>C buses as described in the document *200-307 I/O Board II User Guide and Reference Manual*.

Figure 47. I<sup>2</sup>C Busses on the 80-0062 I/O Board II

#	Name	Description
1	IO EEPROM	Connects to an 24AA02 256-byte EEPROM.
2	IO Thermal	Connects to an LM89CIM thermal monitor supporting on-board ambient temperature sensor and optional external temperature sensor.
3	Backplane	Expansion bus available by connector to J43 on the 80-0063 ConnectBus II Backplane II.

## Availability

The *dpci\_i2c\_bus\_name()* function first appeared in the DirectPCI API version 1.4.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

void show_buses(void)
{
    int i;
    char buf[32]; // should be big enough
    int buses;

    buses = dpci_i2c_numbuses();
    for (i = 0; i < buses; i++)
    {
        if (dpci_i2c_bus_name(i, buf, sizeof(buf)) == 0)
            printf("bus %2d: %s\n", i, buf);
    }
}
```

Further examples can be found in the *i2c\_demo.c* program.

## 2.12.4 I<sup>2</sup>C Bus Access

### Definition

```
#include <dpci_core_api.h>

int dpci_i2c_command(struct dpci_i2c_cmdbuf *cmdbufp);
```

### Parameters

**cmdbufp**      The command buffer into which is placed information about the i2c transfer.

### Description

*Dpci\_i2c\_command()* executes a transfer of data on an I<sup>2</sup>C bus according to the contents of the parameter cmdbufp, which is an instance of **struct dpci\_i2c\_cmdbuf**. The members of **struct dpci\_i2c\_cmdbuf** are as follows:

Figure 48. Declaration of struct dpci\_i2c\_cmdbuf

```
#define MAX_I2C_COMMAND_SEGS 8
struct dpci_i2c_cmdbuf
{
    int          bus;
    int          speed_hz;
    int          options;
    int          attempts;
    int          result;
    unsigned char segment_lengths[MAX_I2C_COMMAND_SEGS];
    unsigned char buffer[1];
};
#define I2C_OPTIONS_NORESTART 0x01
#define I2C_OPTIONS_UNLOCK   0x02
#define I2C_OPTIONS_FORCEACK 0x04
```

The members are described below.

Figure 49. Members of struct dpci\_i2c\_cmdbuf

Member	Definition
bus	The number of the I <sup>2</sup> C bus on which to transfer the data in <i>buffer</i> . See the notes section for further information.
speed_hz	The speed of the clock in Hz. The device driver may ignore this value if it is invalid or the hardware does not support the speed requested.
options	This word is a bit mask formed by inclusive OR'ing of various bits; it is used to request certain behaviours that may be possible during I <sup>2</sup> C transfers. Valid settings are those assigned to C pre-processor macros with the prefix I2C_OPTIONS_. See below for further information.

Member	Definition
attempts	This is the number of attempts to make at a transfer. If this is zero or invalid, a default number (3) will be used.
result	This is the result of the transfer and is returned by the function.
segment_lengths	This array determines how many bytes are in each segment of data to be transferred. The entry after the last segment (if all segments are not used) should be zero.
buffer	This is the first byte of data transfer. See below for a description of how to allocate memory to store data in this buffer.

The result field can have these values:

Figure 50. I<sup>2</sup>C Command Result Values

Member	Definition
I2C_RESULT_SUCCESS	The data transfer succeeded.
I2C_RESULT_INPROGRESS	The data transfer is still in progress.
I2C_RESULT_NOSLAVE	The data transfer failed because the appointed slave did not send an ACK at bit 9 of the addressing phase.
I2C_RESULT_TXNAK	The data transfer failed because the appointed slave returned NAK when the driver was sending it a byte of data.
I2C_RESULT_LOSTARB	The data transfer failed because another I <sup>2</sup> C master device was accessing the bus.
I2C_RESULT_TIMEDOUT	The data transfer was aborted because it took too long to complete.
I2C_RESULT_INTERNALERR	The data transfer failed for some internal reason. Please report a bug!
I2C_RESULT_STUCKBUS	The data transfer failed because of a communications failure on the requested I <sup>2</sup> C bus.
I2C_RESULT_BUSBUSY	The data transfer could not be initiated because another I <sup>2</sup> C master was continually using the bus.
I2C_RESULT_NOBUS	The data transfer could not be initiated because the requested bus number does not exist.
I2C_RESULT_INVALIDBUF	The data transfer could not be initiated because the command or read/write buffer is somehow invalid.

The options valid for the *options* field are:

Figure 51. I<sup>2</sup>C Command Options

Member	Definition
I2C_OPTIONS_NORESTART	Where the I <sup>2</sup> C hardware controller supports I <sup>2</sup> C bus restarts, restarts are disabled and a start condition is always preceded by a valid start condition.
I2C_OPTIONS_UNLOCK	In some cases, the I <sup>2</sup> C bus may become stuck (if for example, a too-fast bus-speed is used to access a slow peripheral). Setting this option causes the driver to attempt to recover the bus and unlock the stuck state. Note that ALL other fields in the command are ignored when this option is set.
I2C_OPTIONS_FORCEACK	Always acknowledge the last byte of data received from a slave; ordinarily a NAK is sent on bit 9 after the final byte is received.

Preparation of the data buffer and setting of the segment lengths is crucial to correct operation of I<sup>2</sup>C transfers effected using this function. Here is how one should set up the *dpci\_i2c\_cmdbuf* structure in order to read 4 bytes from offset 0x1A20 in an Atmel AT24C64 EEPROM at slave address 0xA4. The example code at the end of this section shows how to use to do this.

Firstly, calculate the data to fit in the buffer. A 4-byte read from the EEPROM described requires 8 bytes to be transferred in two segments (each segment begins with an I<sup>2</sup>C *start* condition and ends with a *stop* condition). In the first segment are the slave address (1 byte) and the EEPROM memory offset (two bytes); in the second segment are the slave address (with bit 0 set) and then the four bytes of data (which the EEPROM returns).

Note that slave addresses should be inserted as they would be transmitted on the bus: this means that a one-bit left-shift on the address is required in order to fit in the needed R/W setting at bit 0.

Next, memory must be allocated for data. The memory should follow immediately after the *buffer* member of an instance of *struct dpci\_i2c\_cmdbuf*. The segment lengths and data buffer arrays may now be populated. (Again, the sample code below shows how to do this.)

Finally, the *dpci\_i2c\_command()* function may be invoked.

## Return Value

- |    |   |
|----|---|
| 0  | The data transfer succeeded.                    |
| -1 | Failure performing the requested data transfer. |

If the data transfer succeeded then the *result* field of the *dpci\_i2c\_cmdbuf* structure used will be set to I2C\_RESULT\_SUCCESS. In all other cases, the *result* field will be set to one of the other result codes listed above.

## Notes

This function is not for the faint-hearted; users are advised to gain an understanding of I<sup>2</sup>C bus peripherals and protocols before use.

This function is only usable on DPX-C series boards used in conjunction with the model 80-0062 I/O board and model 80-0063 Backplane.

You should be careful on multi-drop I<sup>2</sup>C buses not to set the speed of the bus to operate any higher than the maximum supported speed of the *slowest* device on the bus. Operating beyond that limit may cause data corruption and an unrecoverable bus lockup which may only be reset with a full hardware reset or power cycle.

The Innocore DPX-series mainboards named above all have one I<sup>2</sup>C bus available which may be used to access the on-board EEPROM and, in some cases, peripherals available on the backplane. (Refer to your mainboard manual for full details.) Where available, this bus is always numbered 0.

The Innocore 80-0062 I/O Board II supplies additional I<sup>2</sup>C busses:

## Availability

The *dpci\_i2c\_command()* function first appeared in the DirectPCI API version 1.3.0.

## Example

This example uses the access scenario described above.

```
#include <stdio.h>
#include <dpci_core_api.h>

...
#define BUS      0      // main board bus.
#define DATA_SIZE 8      // Data transfer size to read 1 byte
#define SLAVE    0xa0    // eeprom on main-board bus.
#define OFFSET   0x1a20 // where in eeprom's memory

struct dpci_i2c_cmdbuf *cmdbuf;

// Allocated one byte less since the cmdbuf contains the first
// byte of the buffer.
//
cmdbuf = malloc(sizeof(*buf) + DATA_SIZE - 1);
memset(cmdbuf, sizeof(*cmdbuf), 0);
cmdbuf->bus = BUS;

// Initialise segment lengths
//
cmdbuf->segment_lengths[0] = 3;
cmdbuf->segment_lengths[1] = 5;
cmdbuf->segment_lengths[2] = 0; // marks end of segment list

// Initialise segment 0
//
cmdbuf->buffer[0] = SLAVE;
```

```
cmdbuf->buffer[1] = OFFSET & 0xff;
cmdbuf->buffer[2] = OFFSET > 8;

// Initialise segment 1. We only initialise the first byte - the
// slave returns the last four bytes. Set the R/W bit (bit 0)
// to indicate a read transfer (master receives, slave
// transmits).
//
cmdbuf->buffer[3] = SLAVE | 1;

// Now call the function. Do error handling...
//
result = dpci_i2c_command(cmdbuf);
if (result != 0)
{
    fprintf(stderr,
            "Error in I2C transfer: %s\n",
            dpci_i2c_error_string(cmdbuf->result));
    return;
}

printf("data returned is: %02x %02x %02x %02x\n",
       buffer[4],
       buffer[5],
       buffer[6],
       buffer[7]);
```

Further examples can be found in the *i2c\_demo.c* program.

## 2.12.5 TRANSLATE I<sup>2</sup>C ERROR CODE

### Definitions

```
#include <dpci_core_api.h>

const char *dpci_i2c_error_string(int code)
```

### Parameters

code            The error code to translate.

### Description

*Dpci\_i2c\_error\_string()* returns a textual ASCII equivalent to the error code given in the parameter *code*. This code is the value returned in the *result* field of structures used by *dpci\_i2c\_command()*, *dpci\_i2c\_read()* or *dpci\_i2c\_write()*, or the value returned by the function *dpci\_i2c\_last\_error()*.

### Return Value

error string    A valid pointer to a NUL-terminated ASCII string is always returned, even if the *code* parameter does not correspond to a known I<sup>2</sup>C error code.

### Notes

This function is only usable where I<sup>2</sup>C bus facilities are available via DirectPCI hardware.

### Availability

The *dpci\_i2c\_error\_string()* function first appeared in the DirectPCI API version 1.3.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

void docmd(struct dpci_i2c_cmdbuf *buf)
{
    if (dpci_i2c_command(buf) != 0)
    {
        fprintf(stderr,
                "I2C command to bus %d failed: i2c error %s\n",
                buf->bus,
                dpci_i2c_error_string(buf->result));
    }
}
```

Further examples can be found in the *i2c\_demo.c* program.

## 2.12.6 RETURN LAST I<sup>2</sup>C ERROR CODE

### Definitions

```
#include <dpci_core_api.h>

int dpci_i2c_last_error(void)
```

### Parameters

None

### Description

The *dpci\_i2c\_last\_error()* returns the I<sup>2</sup>C error code from the last I<sup>2</sup>C command executed whether that was for an eeprom command, I<sup>2</sup>C read/write command or I<sup>2</sup>C buffer command.

### Return Value

Error code	The last I <sup>2</sup> C result (error) code.
------------	--

### Notes

The result code is saved in the function *dpci\_i2c\_command* so anything calling that function will result in the error code changing.

This function is only usable where I<sup>2</sup>C bus facilities are available via DirectPCI hardware.

### Availability

The *dpci\_i2c\_last\_error()* function first appeared in the DirectPCI API version 1.3.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

void print_i2c_error(void)
{
    int error = dpci_i2c_last_error();

    fprintf(stderr,
            "i2c error %s\n",
            dpci_i2c_error_string(error));
}
```

Further examples can be found in the *i2c\_demo.c* program.

## 2.12.7 RETURN LAST I<sup>2</sup>C ERROR STRING

### Definitions

```
#include <dpci_core_api.h>

Const char* dpci_i2c_last_error_string (void)
```

### Parameters

None

### Description

The *dpci\_i2c\_last\_error\_string()* returns the description of the last error from the last I<sup>2</sup>C command executed whether that was for an eeprom command, I<sup>2</sup>C read/write command or I<sup>2</sup>C buffer command.

### Return Value

error string	A valid pointer to a NUL-terminated ASCII string is always returned, even if the <i>code</i> parameter does not correspond to a known I <sup>2</sup> C error code.
--------------	--

### Notes

This function is only usable where I<sup>2</sup>C bus facilities are available via DirectPCI hardware.

### Availability

The *dpci\_i2c\_last\_error\_string()* function first appeared in the DirectPCI API version 3.0.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

void print_i2c_error(void)
{
    fprintf(stderr,
            "i2c_error %s\n",
            dpci_i2c_last_error_string());
```

## 2.12.8 GENERIC I<sup>2</sup>C READ ACCESS

### Definitions

```
#include <dpci_core_api.h>

int dpci_i2c_read(struct dpci_i2c_rdbuf *rdbuf)
```

### Parameters

`rdbuf` The buffer into which is placed information about the I<sup>2</sup>C data read transfer.

### Description

`Dpci_i2c_read()` executes a read transfer of data on an I<sup>2</sup>C bus according to the contents of the parameter `rdbuf`, which is an instance of struct `dpci_i2c_rdbuf`. The members of struct `dpci_i2c_rdbuf` are as follow.

Figure 52. Declaration of struct `dpci_i2c_rdbuf`

```
struct dpci_i2c_rdbuf
{
    int          bus;
    int          slave;
    int          speed_hz;
    int          options;
    int          attempts;
    int          result;
    int          pagesize;
    int          offset;
    int          count;
    char        *buf;
};

#define I2C_OPTIONS_NORESTART      0x01
#define I2C_OPTIONS_UNLOCK         0x02
#define I2C_OPTIONS_FORCEACK       0x04
#define I2C_OPTIONS_ADDRMASK       0x18
#define I2C_OPTIONS_1BYTEADDR      0x00
#define I2C_OPTIONS_11BITADDR      0x08
#define I2C_OPTIONS_2BYTEADDR      0x10
```

Its members are described below.

Figure 53. Members of struct `dpci_i2c_rdbuf`

Member	Definition
<code>bus</code>	The number of the I <sup>2</sup> C bus on which to transfer the data in <code>buffer</code> .
<code>speed_hz</code>	The speed of the clock in Hz. The device driver may ignore this value if it is invalid or the hardware does not support the speed requested.

Member	Definition
options	This word is a bit mask formed by inclusive OR'ing of various bits; it is used to request certain behaviours that may be possible during I <sup>2</sup> C transfers. Valid settings are those assigned to C pre-processor macros with the prefix I2C_OPTIONS_.
attempts	This is the number of attempts to make at a transfer. If this is zero or invalid, a default number (3) will be used.
result	This is the result of the transfer and is modified by the function. See the description of <i>dpci_i2c_command()</i> for details of the different result codes.
pagesize	The size of pages on this device. This determines how many bytes can be read or written in a single transfer. Consult the data sheet for the intended peripheral to determine how to set this field. If left zero, then a page-size of 1 byte is assumed.
offset	This is the offset in the slave's register/memory space of the first location to be accessed. Subsequent accesses are always one location on from the previous access location.
count	This determines how many bytes are to be read from the slave.
buf	This is a pointer to the memory where the first byte of read data will be copied.

The options valid for the *options* field are listed here and are in addition to those available to the *dpci\_i2c\_command* function. The main options that are specific to this function are related to how the *offset* field is transmitted to the chosen slave.

Figure 54. Options for struct dpci\_i2c\_rwbuf.options

Member	Definition
I2C_OPTIONS_1BYTEADDR	The offset is transmitted as the first byte following the slave address. Only the 8 LSBs of <i>offset</i> are used.
I2C_OPTIONS_11BITADDR	The offset is transmitted partially in the slave address and partially in the first byte following the slave address. Bits 8-10 of the <i>offset</i> are transposed to bits 0-2 of the slave address and the 8 LSBs of <i>offset</i> are transmitted immediately after the slave address. This mode is required when communicating with some EEPROMs with sizes in the range 256..2048 bytes, such as the Microchip 24AA0x range.
I2C_OPTIONS_2BYTEADDR	The offset is transmitted as the first two bytes following the slave address. Only the 16 LSBs of <i>offset</i> are used.

Using *dpci\_i2c\_read()* offers a much simpler way of reading multiple bytes from a peripheral's memory space than is achievable using *dpci\_i2c\_command()*. Instead of requiring a complete transfer buffer to be formulated as per *dpci\_i2c\_command()*, *dpci\_i2c\_read()* instead merely needs enough information about the location and characteristics of the slave and where the data should be copied in the caller's memory space.

*Dpci\_i2c\_read()* attempts to arrange the data reception as efficiently as possible, automatically adjusting the individual data transfers according to the page-size and addressing structure of the chosen peripheral as defined by the caller.

## Return Value

- |    |   |
|----|---|
| >0 | The read access returned the actually read number of bytes. This may be less than the requested number if, for some reason, the transfer encountered a problem mid way. |
| 0  | The read access returned no bytes.  |
| -1 | The read access failed. The reason for the failure can be determined from the system error number and the <i>result</i> member of the read/write buffer.                |

## Notes

After the call, all members of the structure at *rdbufp* are unchanged except for *result*.

The effects of accessing an I<sup>2</sup>C device using the wrong speed, the wrong page-size or the wrong memory address length can be serious:

Figure 55. Known problems with I<sup>2</sup>C Addressing and Bus Speeds

Cause	Effect
Using the wrong-device speed	<p>Unrecoverable bus-lockup requiring hardware reset or system power cycle.</p> <p>Data corruption in communication with all devices.</p> <p>See the note on speed for <i>dpci_i2c_command()</i>.</p>
Using the wrong memory address size	<p>If an 8-bit or 11-bit memory word address is required but a 16-bit word is used then data corruption can occur at the 8-bit address given by the first byte of address transmitted.</p> <p>If a 16-bit word is required but an 8-bit word is used then data corruption can occur because the first byte of data to write is interpreted as part of the data address.</p> <p>Using 11-bit addresses where 16-bit addresses are required may cause the I<sup>2</sup>C access to fail because the EEPROM does not respond to the modified I<sup>2</sup>C devices addresses implicit in 11-bit addresses.</p>

Cause	Effect
Using the wrong page-size	<p>If the page size is incorrectly set larger than the device supports:</p> <p>On reading, invalid data might be returned from an EEPROM if the device does not automatically move to the next page of memory.</p> <p>On writing, the write access may fail completely or partially with an error indicating the device is unresponsive or returns a NAK before the I<sup>2</sup>C controller transmits the last byte of data to it. In other cases, data corrupt may occur if the device address counter wraps around to the beginning of the same page and not onto the next page.</p>

## Availability

The *dpci\_i2c\_read()* function first appeared in the DirectPCI API version 1.3.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
#define PAGESIZE    32      // AT24C64 EEPROM with 32-byte pages
#define BUS        0       // main-board bus.
#define SLAVE      0xa0    // eeprom on main-board bus.
#define OFFSET     0x1a20  // where in eeprom's memory

struct dpci_i2c_rwbuf rdbuf;
char data[64];

// Set up the buffer.
//
memset(rdbuf, 0, sizeof(rdbuf));
rdbuf.bus      = BUS;
rdbuf.slave    = SLAVE;
rdbuf.pagesize = PAGESIZE;
rdbuf.offset   = OFFSET;
rdbuf.options  = I2C_OPTIONS_2BYTEADDR;
rdbuf.count    = sizeof(data);
rdbuf.buf      = data;

// Now call the function. Do error handling...
//
result = dpci_i2c_read(rdbuf);
if (result != sizeof(data))
{
    fprintf(stderr,
            "Error in I2C transfer: %s\n",
            dpci_i2c_error_string(cmdbuf->result));
    return;
}

printf("data returned is: %02x %02x %02x %02x ... \n",
       data[0],
       data[1],
       data[2],
       data[3]);
...
```

## 2.12.9 GENERIC I<sup>2</sup>C WRITE ACCESS

### Definitions

```
#include <dpci_core_api.h>

int dpci_i2c_write(struct dpci_i2c_rdbuf *rdbuf)
```

### Parameters

`rdbuf` The buffer into which is placed information about the I<sup>2</sup>C data write transfer.

### Description

`Dpci_i2c_write()` executes a write transfer of data on an I<sup>2</sup>C bus according to the contents of the parameter `rdbuf`, which is an instance of `struct dpci_i2c_rdbuf`. The members of `struct dpci_i2c_rdbuf` are as documented in [dpci\\_i2c\\_read\(\)](#) on p208.

Using `dpci_i2c_write()` offers a much simpler way of writing multiple bytes to a peripheral's memory space than is achievable using `dpci_i2c_command()`. Instead of requiring a complete transfer buffer to be formulated as per `dpci_i2c_command()`, `dpci_i2c_write()` instead merely needs enough information about the location and characteristics of the slave and where the data should be obtained from the caller's memory space.

`Dpci_i2c_write()` attempts to arrange the data reception as efficiently as possible, automatically adjusting the individual data transfers according to the page-size and addressing structure of the chosen peripheral as defined by the caller.

### Return Value

- |    |  |
|----|--|
| >0 | The write access returns the actually written number of bytes. This may be less than the requested number if, for some reason, the transfer encountered a problem mid way. |
| 0  | The write access wrote no bytes.   |
| -1 | The write access failed. The reason for the failure can be determined from the system error number and the <code>result</code> member of the read/write buffer.            |

### Notes

After the call, all members of the structure at `rdbuf` are unchanged except for `result`.

### Availability

The `dpci_i2c_write()` function first appeared in the DirectPCI API version 1.3.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...

#define PAGESIZE    32      // DS1307 RTC/RAM
#define BUS         2       // backplane bus on 80-0063 b/p II
#define SLAVE       0xd0    // eeprom on main-board bus.
#define OFFSET      0x0     // where in DS1307's memory

struct dpci_i2c_rwbuf rdbuf;
char data[] = {0x06, 0x03, 0x04, 0x23, 0x59, 0x59};

// Set up the buffer.
// 
memset(rdbuf, 0, sizeof(rdbuf));
rdbuf.bus      = BUS;
rdbuf.slave    = SLAVE;
rdbuf.pagesize = PAGESIZE;
rdbuf.offset   = OFFSET;
rdbuf.options  = I2C_OPTIONS_1BYTEADDR;
rdbuf.count    = sizeof(data);
rdbuf.buf      = data;

// Now call the function. Do error handling...
// 
result = dpci_i2c_write(rdbuf);
if (result != sizeof(data))
{
    fprintf(stderr,
            "Error in I2C transfer: %s\n",
            dpci_i2c_error_string(cmdbuf->result));
    return;
}

printf("data returned is: %02x %02x %02x %02x ... \n",
       data[0],
       data[1],
       data[2],
       data[3]);
...
```

## 2.12.10 8-BIT I<sup>2</sup>C READ ACCESS

### Definitions

```
#include <dpci_core_api.h>

int dpci_i2c_read8(int bus,
                    int slave,
                    unsigned int offset,
                    unsigned char *pvalue)
```

### Parameters

<b>bus</b>	The number of the I <sup>2</sup> C bus on which <i>slave</i> resides.
<b>slave</b>	The address of the <i>slave</i> from which data should be read.
<b>offset</b>	The offset in the slave's address space from which to read data. This must be in the range 0..255 (0x00...0xff).
<b>pvalue</b>	Where to place the data returned.

### Description

*Dpci\_i2c\_read8()* executes a read transfer of one byte of data on an I<sup>2</sup>C bus according to the parameters *bus*, *slave* and *offset*. The slave is expected to support a 1-byte encoding of the *offset* parameter, which limits that parameter to the range indicated above.

### Return Value

0	The read access succeeded.
-1	The read access failed. The reason for the failure can be determined from the system error number and the I <sup>2</sup> C error number obtainable using <i>dpci_i2c_last_error()</i> .

### Availability

The *dpci\_i2c\_read8()* function first appeared in the DirectPCI API version 1.3.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
#define BUS      2      // main-board bus.
#define SLAVE    0x58    // some thermal device on B/P II bus.
#define OFFSET   0x13    // where in eeprom's memory

// Call the function. Do error handling...
// 
unsigned char data;
result = dpci_i2c_read8(BUS, SLAVE, OFFSET, &data);
if (result == -1)
{
    fprintf(stderr,
            "Error in I2C transfer: %s\n",
            dpci_i2c_error_string(dpci_i2c_last_error()));
    return;
}

printf("data returned is: %02x\n", data);
```

## 2.12.11 8-BIT I<sup>2</sup>C WRITE ACCESS

### Definitions

```
#include <dpci_core_api.h>

int dpci_i2c_write8(int bus,
                     int slave,
                     unsigned int offset,
                     unsigned char value)
```

### Parameters

<b>bus</b>	The number of the I <sup>2</sup> C bus on which <i>slave</i> resides.
<b>slave</b>	The address of the <i>slave</i> to which data should be written.
<b>offset</b>	The offset in the slave's address space to which to write data. This must be in the range 0..255 (0x00...0xff).
<b>value</b>	The data to write

### Description

*Dpci\_i2c\_write8()* executes a write transfer of one byte of data on an I<sup>2</sup>C bus according to the parameters *bus*, *slave* and *offset*. The slave is expected to support a 1-byte encoding of the *offset* parameter, which limits that parameter to the range indicated above.

### Return Value

0	The write access succeeded.
-1	The write access failed. The reason for the failure can be determined from the system error number and the I <sup>2</sup> C error number obtainable using <i>dpci_i2c_last_error()</i> .

### Availability

The *dpci\_i2c\_write8()* function first appeared in the DirectPCI API version 1.3.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...

#define BUS      2      // main-board bus.
#define SLAVE    0x58    // some thermal device on B/P II bus.
#define OFFSET   0x13    // where in eeprom's memory

// Call the function. Do error handling...
//
unsigned char data = 33;
result = dpci_i2c_write8(BUS, SLAVE, OFFSET, data);
if (result == -1)
{
    fprintf(stderr,
            "Error in I2C transfer: %s\n",
            dpci_i2c_error_string(dpci_i2c_last_error()));
    return;
}
```

## 2.13 TEMPERATURE SENSOR API (I/O BOARD II)

### 2.13.1 TEMPERATURE SENSOR INITIALISATION

#### Definitions

```
#include <dpci_core_api.h>

int dpci_ts_init()
```

#### Parameters

None

#### Description

*Dpci\_ts\_init()* initializes the LM89 on a mode 80-0062 I/O Board II. It does this simply by setting the device's configuration register to disable alerts and critical interrupts and enable normal running. The function does not change any other registers.

#### Return Value

- |    |  |
|----|--|
| 0  | The configuration succeeded.   |
| -1 | The configuration access failed. The reason for the failure can be determined from the system error number and the I <sup>2</sup> C error number obtainable using <i>dpci_i2c_last_error()</i> . |

#### Notes

This function is the only way to reset the ALERTN flag in the LM89 configuration register other than changing it using *dpci\_i2c\_write8()*.

#### Availability

The *dpci\_ts\_init()* function first appeared in the DirectPCI API version 1.4.0.

#### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

// Call the function.  Do error handling...
//
if (dpci_ts_init() == -1)
{
    fprintf(stderr,
            "Error in setting up LM89: %s\n",
            dpci_i2c_error_string(dpci_i2c_last_error()));
    return;
}
```

A further example of this can be found in the demonstration *ts\_demo.c* program.

## 2.13.2 GET NUMBER OF TEMPERATURE SENSORS

### Definitions

```
#include <dpci_core_api.h>

int dpci_ts_numsensors();
```

### Parameters

None.

### Description

*Dpci\_ts\_numsensors* allows the programmer to retrieve the number of temperature sensors supplied by the system's DirectPCI hardware.

### Return Value

- |     |   |
|-----|---|
| >=0 | The number of temperature sensors supplied by DirectPCI hardware. |
| -1  | An error occurred while obtaining the number of buses.            |

### Notes

See the description of *dpci\_ts\_sensor\_name()* for more information on the buses that are available on DPX-series mainboards.

### Availability

The *dpci\_ts\_numsensors()* function first appeared in the DirectPCI API version 1.4.0.

### Example

See the example code for *dpci\_ts\_sensor\_name()*. Further examples can be found in the *ts\_demo.c* program.

### 2.13.3 GET BUS NUMBER FOR A NAMED TEMPERATURE SENSOR

#### Definitions

```
#include <dpci_core_api.h>

int dpci_ts_sensor_number(char *name);
```

#### Parameters

**name** The name of the sensor whose logical number is to be obtained.

#### Description

*Dpci\_ts\_sensor\_number()* allows the programmer to retrieve the number of a temperature sensor whose name is known.

#### Return Value

>=0	The number of the sensor whose name is in <i>name</i> .
-1	An error occurred while obtaining the sensor's name.

#### Notes

Note that name comparison is presently case-sensitive.

See the description of *dpci\_ts\_sensor\_name()* for more information on the sensors that are available on DPX-series mainboards.

#### Availability

The function *dpci\_ts\_sensor\_number()* first appeared in the DirectPCI API version 1.4.0.

#### Example

```
#include <dpci_core_api.h>
#include <stdio.h>

int find_sensor_or_die(char *name)
{
    int sensor;

    sensor = dpci_ts_sensor_number(name);
    if (sensor == -1)
    {
        fprintf(stderr,
                "Sensor named %s is not available.\n", name);
        exit(1);
    }
    printf("Sensor number for %s is %d.\n", name, sensor);
}
```

Further examples can be found in the *ts\_demo.c* program.

## 2.13.4 GET TEMPERATURE SENSOR NAME

### Definitions

```
#include <dpci_core_api.h>

int dpci_ts_sensor_name(int sensor, char *buf, int bufsiz);
```

### Parameters

sensor	The sensor number whose name is to be returned
buf	The buffer into which the sensor name should be returned.
bufsiz	The size of the buffer <i>buf</i> .

### Description

*Dpci\_ts\_sensor\_name* allows the programmer to retrieve the logical name for a given temperature sensor number. The name is returned in the buffer pointed to by *buf*. At most *bufsiz* bytes are copied to *buf*, including a NUL-terminator. Generally the names assigned to temperature sensors are less than 16 characters long.

### Return Value

0	The name was returned correctly. It may have been truncated if <i>bufsiz</i> was too small.
-1	An error occurred while obtaining the sensors's name, for example if the sensor number is invalid.

### Notes

This function is only usable where I<sup>2</sup>C bus facilities are available via DirectPCI hardware. The following table shows the I<sup>2</sup>C bus support for various models of DPX-series mainboards.

The 80-0062 I/O Board supplies the following temperature sensors.

Figure 56. Temperature Sensors on the 80-0062 I/O Board II

#	Name	Description
0	onboard	The temperature sensor built into the LM89CIM.
1	remote	The temperature sensor connected via J4.

### Availability

The *dpci\_ts\_sensor\_name()* function first appeared in the DirectPCI API version 1.4.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

void show_sensors(void)
{
    int i;
    char buf[32]; // should be big enough
    int sensors;

    sensors = dpci_ts_numensors();
    for (i = 0; i < sensors; i++)
    {
        if (dpci_ts_sensor_name(i, buf, sizeof(buf)) == 0)
            printf("Sensor %2d: %s\n", i, buf);
    }
}
```

Further examples can be found in the *ts\_demo.c* program.

## 2.13.5 READ TEMPERATURE SENSOR

### Definitions

```
#include <dpci_core_api.h>

int dpci_ts_getsensor(int sensor,
                      int *tempp,
                      int *connectedp);

#define IOBD2_SENSOR_ONBOARD      0x00
#define IOBD2_SENSOR_REMOTE       0x01
```

### Parameters

sensor	The sensor to read
tempp	Where to return the temperature reading for the chosen sensor.
connectedp	Where to return a Boolean value indicating whether the chosen sensor is connected.

### Description

*Dpci\_ts\_getsensor()* attempts to read the temperature sensor on the LM89 chip of the model 80-0062 I/O Board II.

Any of the pointers *tempp* and *connectedp* may be NULL, in which case that parameter is ignored.

### Return Value

0	The call succeeded.
-1	The call access failed. The reason for the failure can be determined from the system error number and the I2C error number obtainable using <i>dpci_i2c_last_error()</i> .

### Notes

The onboard sensor (IOBD2\_SENSOR\_ONBOARD) is always connected so long as a suitable I/O board is connected.

A suitable diode and transistor must be connected across J4 of the I/O board in order to sense temperature using the remote sensor (IOBD2\_SENSOR\_REMOTE). If one is connected then 1 is written to the location (if any) pointed to by the *connectedp* parameter; otherwise zero is written.

## Availability

This function is only usable in conjunction with the model 80-0062 I/O board and 80-0063 backplane.

The *dpci\_ts\_getsensor()* function first appeared in the DirectPCI API version 1.4.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
int local, remote, connected;

// Call the function.  Do error handling...
//if (dpci_ts_getsensor(IOBD2_SENSOR_ONBOARD, &local, NULL) == -1)
//{
//    fprintf(stderr,
//            "Error in reading LM89 onboard sensor: %s\n",
//            dpci_i2c_error_string(dpci_i2c_last_error()));
//    return;
//}
//if (dpci_ts_getsensor(IOBD2_SENSOR_REMOTE, &remote, &connected) == -1)
//{
//    fprintf(stderr,
//            "Error in reading LM89 remote sensor: %s\n",
//            dpci_i2c_error_string(dpci_i2c_last_error()));
//    return;
}
printf("Local temperature: %d deg.C\n", local);
if (connected)
    printf("Remote temperature: %d deg.C\n", remote);
```

A further example of this can be found in the demonstration *ts\_demo.c* program.

## 2.13.6 CONFIGURE TEMPERATURE SENSOR

### Definitions

```
#include <dpci_core_api.h>

int dpci_ts_configure(int sensor,
                      int high,
                      int low,
                      int crit,
                      int offs,
                      int alarm);
#define IOBD2_SENSOR_ONBOARD      0x00
#define IOBD2_SENSOR_REMOTE       0x01

#define TS_ALARM_NONE            0x00      /* Do not report any alarms */
#define TS_ALARM_CRIT             0x01      /* Report for critical levels */
#define TS_ALARM_HIGHLO           0x02      /* Report for high/low levels */
```

### Parameters

<b>sensor</b>	The sensor number to configure.
<b>high</b>	The temperature level at or above which an alert is raised
<b>low</b>	The temperature level at or below an alert is raised
<b>crit</b>	The temperature at or above which a critical alert is raised.
<b>offs</b>	For the remote temperature sensor only, the offset (in degrees Celsius) applied to temperature readings.
<b>alarm</b>	The type of alarm to configure for the sensor

### Description

*Dpci\_ts\_configure()* attempts to configure a single temperature sensor given by **sensor** on the LM89 chip of the model 80-0062 I/O Board II.

The parameters passed are programmed in to the appropriate registers on the LM89. The offset parameter is only available for the remote temperature sensor and any value supplied is ignored if the sensor to be programmed is the local (on-board) sensor.

### Return Value

0	The configuration succeeded.
-1	The configuration access failed. The reason for the failure can be determined from the system error number and the I2C error number

obtainable using *dpci\_i2c\_last\_error()*.

## Notes

The alarm generating capability of the LM89 is limited as follows: the ability to generate an alarm for a given sensor reports a violation of either the low or high levels for both the on-board and remote sensors, i.e. if either sensor's temperature is below its low or above its high configured level. This limitation does not apply to the critical level; however, if either sensor is configured to give an alarm at either the high or the low levels then both sensors will. To work around this, set the high temperature and critical temperature level to be the same for sensors where you don't want a high level of temperature to cause an alarm.

## Availability

This function is only usable in conjunction with the model 80-0062 I/O board and 80-0063 backplane.

The *dpci\_ts\_configure()* function first appeared in the DirectPCI API version 1.4.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
// Call the function.  Do error handling...
//if (dpci_ts_configure(IOBD2_SENSOR_ONBOARD,
//                      20, // high
//                      10, // low
//                      30, // critical
//                      0, // offset (unused for onboard) sensor
//                      TS_ALARM_CRIT) == -1)
//{
//    fprintf(stderr,
//            "Error configuring on-board sensor: %s\n",
//            dpci_i2c_error_string(dpci_i2c_last_error()));
//    return;
//}
```

A further example of this can be found in the demonstration *ts\_demo.c* program.

## 2.13.7 READ TEMPERATURE SENSOR CONFIGURATION

### Definitions

```
#include <dpci_core_api.h>

int dpci_ts_getconfig(int *sensorp,
                      int *highp,
                      int *lowlp,
                      int *critp,
                      int *offsp,
                      int *alarmp);
#define IOBD2_SENSOR_ONBOARD      0x00
#define IOBD2_SENSOR_REMOTE       0x01

#define TS_ALARM_NONE            0x00      /* Do not report any alarms */
#define TS_ALARM_CRIT             0x01      /* Report for critical levels */
#define TS_ALARM_HIGHLO           0x02      /* Report for high/low levels */
```

### Parameters

sensor	The sensor number to configure.
highp	The temperature level at or above which an alert is raised
lowlp	The temperature level at or below which an alert is raised
critp	The critical temperature at or above which an critical alert is raised.
offsp	For the remote temperature sensor only, the offset (in degrees Celsius) applied to temperature readings.
alarmp	The type of alarm to configure for the sensor

### Description

*Dpci\_ts\_getconfig()* returns the current configuration settings for the given sensor *sensor*.

The offset parameter is only available for the remote temperature; the parameter is set to zero if the sensor is the local (on-board) sensor.

Any of the parameter pointers may be NULL, in which case no data is written for that parameter; however, the appropriate register on the LM89 might still be accessed.

### Return Value

0	The access succeeded.
-1	The access failed. The reason for the failure can be determined from the system error number and the I2C error number obtainable using

*dpci\_i2c\_last\_error()*.

## Notes

Please read the notes section for *dpci\_ts\_configure()* regarding the alarm capability of the LM89. A consequence of the LM89's design is that the alarm type returned by *dpci\_ts\_getconfig()* will be TS\_ALARM\_HIGHLO if that sensor was configured with TS\_ALARM\_NONE and the other sensor was configured with TS\_ALARM\_HIGHLO.

## Availability

This function is only usable in conjunction with the model 80-0062 I/O board and 80-0063 backplane.

The *dpci\_ts\_getconfig()* function first appeared in the DirectPCI API version 1.4.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
int high, low, crit, alarm, offs;
// Call the function.  Do error handling...
//if (dpci_ts_getconfig(IOBD2_SENSOR_REMOTE,
//                      &high,
//                      &low,
//                      &crit,
//                      &offs,
//                      &alarm) == -1)
//{
//    fprintf(stderr,
//            "Error configuring on-board sensor: %s\n",
//            dpci_i2c_error_string(dpci_i2c_last_error()));
//    return;
//}
printf("Low temperature setting: %d\n", low);
printf("High temperature setting: %d\n", high);
printf("Critical temperature setting: %d\n", crit);
printf("Offs temperature setting: %d\n", offs);
printf("Alarm type: %d\n", alarm);
```

A further example of this can be found in the demonstration *ts\_demo.c* program.

## 2.13.8 CHECK TEMPERATURE SENSORS' STATES

### Definitions

```
#include <dpci_core_api.h>

int dpci_ts_check_state(int sensor_mask ,...);
#define IOBD2_SENSOR_ONBOARD          0x00
#define IOBD2_SENSOR_REMOTE           0x01
#define IOBD2_SENSOR_MASK_ONBOARD    (1 << IOBD2_SENSOR_ONBOARD)
#define IOBD2_SENSOR_MASK_REMOTE     (1 << IOBD2_SENSOR_REMOTE)
#define IOBD2_SENSOR_MASK_ALL \      (IOBD2_SENSOR_MASK_ONBOARD | IOBD2_SENSOR_MASK_REMOTE)

#define TS_STATE_NORMAL   0x00
#define TS_STATE_LOW      0x01
#define TS_STATE_HIGH     0x02
#define TS_STATE_CRIT     0x03
```

### Parameters

`sensor_mask` The mask of sensors for which to report temperature status.  
`...` Further arguments detailing where to return sensor state.

### Description

`Dpci_ts_check_state()` checks the states of the temperature sensors determined by the bit mask `sensor_mask`. Because a mask and a variable list of arguments are used, it is possible to obtain state information for multiple sensors at once.

One additional pointer to an integer must be provided for each bit set to 1 in `sensor_mask`. The bit mask `sensor_mask` is processed starting with the least significant bit and proceeding toward the most significant bit; additional arguments are processed starting with the first argument for the least significant bit and then subsequent arguments for subsequent bits.

### Return Value

0	The access succeeded.
-1	The access failed. The reason for the failure can be determined from the system error number and the I2C error number obtainable using <code>dpci_i2c_last_error()</code> .

### Notes

Arguments must be provided for each bit that is set in the `sensor_mask` argument.

## Availability

This function is only usable in conjunction with the model 80-0062 I/O board and 80-0063 backplane.

The *dpci\_ts\_check\_state()* function first appeared in the DirectPCI API version 1.4.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
int onboard_state, remote_state;

// Call the function. Do the on-board sensor
// if (dpci_ts_check_state(IOBD2_SENSOR_MASK_ONBOARD,
//                         &onboard_state) == -1)
{
    fprintf(stderr,
            "Error get on-board sensor state: %s\n",
            dpci_i2c_error_string(dpci_i2c_last_error()));
    return;
}

// Or maybe read them both together.
if (dpci_ts_check_state(IOBD2_SENSOR_MASK_ALL,
                        &onboard_state,
                        &remote_state) == -1)
{
    fprintf(stderr,
            "Error get on-board sensor state: %s\n",
            dpci_i2c_error_string(dpci_i2c_last_error()));
    return;
}

printf("On-board sensor state: %d\n", onboard_state);
printf("Remote sensor state: %d\n", remote_state);
```

A further example of this can be found in the demonstration *ts\_demo.c* program.

## 2.13.9 WAIT FOR A TEMPERATURE SENSOR INTERRUPT

### Definitions

```
#include <dpci_core_api.h>

int dpci_ts_wait_alarm(unsigned long timeout);
```

### Parameters

**timeout**      The maximum time to wait in milliseconds for the interrupt before giving up.

### Description

*Dpci\_ts\_wait\_alarm()* waits for a temperature sensor interrupt to occur. The timeout is supplied in milliseconds. At least one sensor ought previously to have been configured to raise an interrupt for some alarm or critical condition.

An interrupt might be flagged by any supported sensor and for any reason: it is not presently possible to wait only for alarm or critical events on only one sensor when both sensors have been configured to raise alarms.

A temperature sensor interrupt that occurs just as the timeout expires will be missed but flagged immediately on a subsequent call to *dpci\_ts\_wait\_alarm()*.

### Return Value

- |    |   |
|----|---|
| 1  | The interrupt occurred before the timeout period expired. |
| 0  | The interrupt did not occur inside the timeout period.    |
| -1 | It was not possible to set up the wait for the interrupt. |

### Notes

At present, only one process or thread may wait for a temperature sensor interrupt concurrently.

Such interrupts are only supported presently when an 80-0062 I/O board is used.

### Availability

This function is only usable in conjunction with the model 80-0062 I/O board and 80-0063 backplane.

The *dpci\_ts\_wait\_alarm()* function first appeared in the DirectPCI API version 1.4.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>

...
#define MY_TIMEOUT_MS      600000 // 10 minutes

int temp;

// Set up the on-board temperature sensor to give us an alert when
// the critical level is reached. For exemplary purposes, get the
// temperature and the set the critical limit 10 degrees Celsius below
// the current reading so that the alarm should trip immediately.
//
if (dpci_ts_getsensor(IOBD2_SENSOR_MASK_REMOTE, &temp, NULL) == -1 ||
    dpci_ts_configure(IOBD2_SENSOR_MASK_REMOTE,
                      temp - 10,
                      temp - 10,
                      temp - 10,
                      TS_ALARM_CRITICAL
                      &onboard_state) == -1)
{
    fprintf(stderr,
            "Couldn't get temperature or configure sensor: %s\n",
            dpci_i2c_error_string(dpci_i2c_last_error()));
    return;
}

// Now wait for something to happen.
//
while (1)
{
    switch(dpci_ts_wait_alarm(MY_TIMEOUT_MS))
    {
        case -1:
            fprintf(stderr,
                    "Error waiting for temperature sensor interrupt: %s\n",
                    strerror(errno));
            break;
        case 0:
            printf("No critical temperature problem in last %d minutes!\n",
                   MY_TIMEOUT_MS / 60000);
            break; // just go around again.
        case 1:
            my_alert("CRITICAL ALERT: MELTDOWN IMMINENT\n");
            shutdown_nuclear_reactor(SHUTDOWN_NOW);
    }
}
```

A further example of this can be found in the demonstration *ts\_demo.c* program.

## 2.13.10 GET VALUE FOR A TEMPERATURE SENSOR STATE-NAME

### Definitions

```
#include <dpci_core_api.h>

int dpci_ts_state_number(char *name);
```

### Parameters

**name** The name of the state whose logical number is to be obtained.

### Description

*Dpci\_ts\_state\_number()* allows the programmer to retrieve the number of temperature sensor state whose name is known. The state values are those TS\_STATE\_ macros list in the description for *dpci\_ts\_check\_state()*.

### Return Value

>=0	The number of the state whose name is in <i>name</i> .
-1	An error occurred while obtaining the bus's name.

### Notes

The following tables show the relation of state numbers to names.

Figure 57. Mapping Temperature-sensor State Values to Text

#	<b>Macro</b>	<b>Text accepted/returned</b>
0	TS_STATE_NORMAL	“normal”
1	TS_STATE_LOW	“low”
2	TS_STATE_HIGH	“high”
3	TS_STATE_CRIT	“critical”

Note that name comparison is presently case-sensitive.

### Availability

The function *dpci\_ts\_state\_number()* first appeared in the DirectPCI API version 1.4.0.

## 2.13.11 GET TEMPERATURE SENSOR STATE-NAME

### Definitions

```
#include <dpci_core_api.h>

const char *dpci_ts_state_name(int state);
```

### Parameters

sensor      The state number whose name is to be returned

### Description

*Dpci\_ts\_state\_name* allows the programmer to retrieve the logical name for a given temperature sensor state.

### Return Value

char \*NULL      The given number is not a valid temperature sensor state

Not-NUL      A valid pointer to a NUL-terminated ASCII character string.

### Availability

The *dpci\_ts\_state\_name()* function first appeared in the DirectPCI API version 1.4.0.

### Example

Examples can be found in the *ts\_demo.c* program.

## 2.14 BATTERY API

### 2.14.1 GET NUMBER OF BATTERIES

#### Definitions

```
#include <dpci_core_api.h>

int dpci_bat_num_batteries(void);
```

#### Parameters

None.

#### Description

*Dpci\_bat\_num\_batteries* allows the programmer to retrieve the number of batteries available on a board.

#### Return Value

-1	An error occurred while retrieving battery count
>=0	Number of batteries

#### Availability

The *dpci\_bat\_num\_batteries()* function first appeared in the DirectPCI API version 2.0.0.

#### Example

Examples can be found in the *batt\_demo.c* program.

## 2.14.2 CHECK BATTERY STATUS

### Definition

```
#include <dpci_core_api.h>

int dpci_bat_read(void);
```

### Parameters

None

### Description

This function returns the status of the system batteries, which are used to power the SRAM chips, IDLP and the optional TPM chip while the system is powered down.

### Return Value

- |     |  |
|-----|--|
| -1  | An error occurred while retrieving the status of the batteries |
| >=0 | The status mask representing the status of all batteries       |

### Notes

Linux drivers prior to v1.2.0 returned the number of batteries whose status was okay.

Use `dpci_bat_get_num_batteries()` to establish how many bit-positions in the mask are valid.

### Availability

As of v2.0.0, this function has been deprecated in favour of the newer `dpci_bat_get_status_mask()`.

The `dpci_bat_read()` function first appeared in the DirectPCI API version 1.0.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
iBatteryStatus = dpci_bat_read();
switch(iBatteryStatus)
{
    case 0:
        printf("Both Batteries FAIL");
        break;
    case 1:
        printf("Battery 1 FAIL");
        break;
    case 2:
        printf("Battery 2 FAIL");
        break;
    case 3:
```

```
    printf("Both Batteries OK");
    break;
default:
    printf("Battery Status Error");
}
```

A further example can be found in the io\_demo.c program.

## 2.14.3 GET BATTERY NAME

### Definitions

```
#include <dpci_core_api.h>

int dpci_bat_name(int bat, char *buf, int bufsiz);
```

### Parameters

bat	The battery number whose name is to be retrieved.
buf	A buffer to return the battery name
bufsiz	Size of the buffer

### Description

*Dpci\_bat\_name* allows the programmer to retrieve the logical name for a given battery number.

### Return Value

-1	An error occurred while retrieving battery name
0	The battery name was retrieved successfully and is available in the buffer "buf"

### Notes

None.

### Availability

The *dpci\_bat\_name* () function first appeared in the DirectPCI API version 2.0.0.

### Example

Examples can be found in the *batt\_demo.c* program.

## 2.14.4 GET BATTERY INDEX NUMBER

### Definitions

```
#include <dpci_core_api.h>

int dpci_bat_number(char *name);
```

### Parameters

name            The name of the battery whose index number is to be retrieved.

### Description

*Dpci\_bat\_number* allows the programmer to retrieve the battery index number corresponding to a given logical name.

### Return Value

-1	An error occurred while retrieving battery index number
>=0	Index number of the battery corresponding to the given logical name.

### Notes

None.

### Availability

The *dpci\_bat\_number()* function first appeared in the DirectPCI API version 2.0.0.

### Example

Examples can be found in the *batt\_demo.c* program.

## 2.14.5 GET BATTERY STATUS MASK

### Definitions

```
#include <dpci_core_api.h>

int dpci_bat_get_status_mask(void);
```

### Parameters

None

### Description

*Dpci\_bat\_get\_status\_mask* allows the programmer to retrieve the status of all available batteries on the board. Starting at the least significant bit, bit positions are populated for each numbered battery known. A 1 indicates the battery is operating correctly; a 0 indicates the battery is not operating correctly. Data in all other bit positions are undefined.

### Return Value

- |     |  |
|-----|--|
| -1  | An error occurred while retrieving the status of the batteries |
| >=0 | The status mask representing the status of all batteries       |

### Notes

You should use *dpci\_bat\_get\_num\_batteries()* to establish how many bit-positions in the mask are valid.

### Availability

The *dpci\_bat\_get\_status\_mask()* function first appeared in the DirectPCI API version 2.0.0.

This function should be used in preference to the now-deprecated *dpci\_bat\_read()* function.

### Example

Examples can be found in the *batt\_demo.c* program.

## 2.14.6 GET BATTERY VOLTAGE LEVEL

### Definitions

```
#include <dpci_core_api.h>

int dpci_bat_get_level(int batt);
```

### Parameters

batt            The index number of the battery whose voltage level is to be retrieved

### Description

*Dpci\_bat\_get\_level* allows the programmer to retrieve the voltage level of a battery.

### Return Value

-1	An error occurred while retrieving battery voltage level
>=0	Battery voltage level in milliVolts.

### Notes

The battery level returned is the result of an immediate reading, however this value will not be tested against any current error level.

### Availability

The *dpci\_bat\_get\_level()* function first appeared in the DirectPCI API version 2.0.0.

### Example

Examples can be found in the *batt\_demo.c* program.

## 2.14.7 GET STATUS OF A BATTERY

### Definitions

```
#include <dpci_core_api.h>

int dpci_bat_get_status(int batt);
```

### Parameters

batt            The index number of the battery whose status is to be returned

### Description

*Dpci\_bat\_get\_status* allows the programmer to retrieve the status of a battery.

### Return Value

0	The battery is faulty
> 0	The battery is ok
-1	An error occurred while retrieving battery status

### Notes

None.

### Availability

The *dpci\_bat\_get\_status()* function first appeared in the DirectPCI API version 2.0.0.

### Example

Examples can be found in the *batt\_demo.c* program.

## 2.14.8 SET BATTERY CHECK PERIOD

### Definitions

```
#include <dpci_core_api.h>
#include <dpci_core_hw.h>

int dpci_bat_set_check_period(int code);

#define BAT_CHECKPERIOD_NONE      0
#define BAT_CHECKPERIOD_1MIN       1
#define BAT_CHECKPERIOD_1HOUR      2
#define BAT_CHECKPERIOD_1DAY        3
#define BAT_CHECKPERIOD_1MONTH     4
```

### Parameters

**code** The period at which the IDLP+ regularly checks the battery status. It can be one of the following options

### Description

*Dpci\_bat\_set\_check\_period* allows the programmer to instruct the IDLP+ to automatically check the battery status at defined periods.

The parameter *code* is selected from one of the following.

Figure 58. List of Battery Check Period Codes

#	<b>Macro</b>	<b>Definition</b>
0	BAT_CHECKPERIOD_NONE	No automatic battery checking, but setting the period to this value will cause an immediate check of the battery levels
1	BAT_CHECKPERIOD_1MIN	1 min: once per minute at zero seconds past the minute
2	BAT_CHECKPERIOD_1HOUR	1 hr: once per hour at zero minutes past the hour;
3	BAT_CHECKPERIOD_1DAY	1 day: once per day at midnight
4	BAT_CHECKPERIOD_1MONTH	1 month: once per month at midnight on the first day of the month

### Return Value

-1 An error occurred while setting the period

0            The check period was set successfully

## Notes

None.

## Availability

The *dpci\_bat\_set\_check\_period()* function first appeared in the DirectPCI API version 2.0.0.

## Example

Examples can be found in the *batt\_demo.c* program.

## 2.14.9 SET BATTERY ERROR LEVEL

### Definitions

```
#include <dpci_core_api.h>

int dpci_bat_set_error_level(int batt, int millivolts);
```

### Parameters

batt	Index number of the battery whose error level is to be set
millivolts	Voltage(in mV) at or below which the battery is considered to be a problem

### Description

*Dpci\_bat\_set\_error\_level* allows the programmer to set the minimum voltage level below which the battery is considered to be a problem.

### Return Value

-1	An error occurred while setting the error voltage level
0	The error voltage level was set successfully

### Notes

None.

### Availability

The *dpci\_bat\_set\_error\_level* () function first appeared in the DirectPCI API version 2.0.0.

### Example

Examples can be found in the *batt\_demo.c* program.

## 2.14.10 GET BATTERY ERROR LEVEL

### Definitions

```
#include <dpci_core_api.h>

int dpci_bat_get_errorlevel(int batt);
```

### Parameters

batt           Index number of the battery whose error level is to be read

### Description

*Dpci\_bat\_get\_errorlevel* allows the programmer read the configured error voltage level of a battery.

### Return Value

-1	An error occurred while reading the error voltage level
int	Configured Error Voltage(in mV) of the battery whose index number is specified

### Notes

This API is supported in firmware version 44 onwards.

### Availability

The *dpci\_bat\_get\_errorlevel* () function first appeared in the DirectPCI API version 2.0.0.

### Example

An example can be found in the *batt\_demo.c* program.

## 2.15 ERROR HANDLING API

The Error handling API is provided for cross-platform development and portability.

### 2.15.1 GET LAST ERROR CODE

#### Definitions

```
#include <dpci_core_api.h>

int dpci_last_os_error_code(void);
```

#### Parameters

None.

#### Description

*dpci\_last\_os\_error\_code* allows the programmer to retrieve the error code of the latest OS error.

#### Return Value

Int	Error code
-----	------------

#### Notes

None.

#### Availability

The *dpci\_last\_os\_error\_code()* function first appeared in the DirectPCI API version 2.0.0.

## 2.15.2 GET LAST ERROR STRING

### Definitions

```
#include <dpci_core_api.h>

char *dpci_last_os_error_string(void);
```

### Parameters

None.

### Description

*dpci\_last\_os\_error\_string* allows the programmer to retrieve the error string of the latest OS error.

### Return Value

string	Error string
--------	--------------

### Notes

None.

### Availability

The *dpci\_last\_os\_error\_string()* function first appeared in the DirectPCI API version 2.0.0.

## 2.15.3 GET ERROR STRING

### Definitions

```
#include <dpci_core_api.h>

char *dpci_os_error_string(int);
```

### Parameters

None.

### Description

*dpci\_os\_error\_string* allows the programmer to retrieve the error string of a given error code.

### Return Value

String	Error string corresponding to the given error code
--------	--

### Notes

None.

### Availability

The *dpci\_os\_error\_string()* function first appeared in the DirectPCI API version 2.0.0.

## 2.16 QUIET-MODE API

### 2.16.1 GET QUIET MODE STATUS

#### Definitions

```
#include <dpci_core_api.h>

int dpci_qm_get(int *pPassthrough, int *pLevel);
```

#### Parameters

pPassthrough	Buffer to hold the quiet mode passthrough (QMPT) status
pLevel	Buffer to hold the quiet mode level (QMLV) status

#### Description

*dpci\_qm\_get()* allows the programmer to read the status of the QMPT and QMLV bits used to configure quiet mode.

#### Return Value

0	Success – The status bits can be read from the pPassthrough and pLevel buffers
< 0	An error occurred while reading the quiet mode status

#### Notes

Quiet Mode can be set to ON/OFF/PASSTHROUGH in software. Setting quiet mode to ON/OFF in software controls the quiet mode setting as per the QMLV bit value. Setting quiet mode to PASSTHROUGH, gives hardware control to the external quiet mode input signal to determine the quiet mode state on the board.

The default state for quiet mode signals on the DPX boards is “OFF-Set”, meaning quiet mode is disabled by default. The result obtained in the buffer is decoded as per the table shown below.

Figure 59. Quiet Mode Fields and Settings

QMPT	QMLV	Quiet Mode status
Zero	Zero	ON – Passthrough
Zero	Non-zero	OFF – Passthrough
Non-zero	Zero	ON – Set

QMPT	QMLV	Quiet Mode status
Non-zero	Non-zero	OFF – Set

Quiet mode is supported in DirectPCI firmware version 71 (or newer) and 81 (or newer).

Quiet mode pass-through mode is effective only on the DPX-S410 and DPX-C705 boards where a suitable input connector is available. Using pass-through mode on other boards may result in undefined behaviour.

On DPX-S410 and C705 boards, enabling quiet mode (whether programmatically or via the pass-trough mechanism) disables the digital outputs and disables audio outputs.

On other S and E series boards, enabling quiet mode merely disables the digital outputs alone, not the audio output.

## Availability

The *dpci\_qm\_get()* function first appeared in the DirectPCI API version 2.0.0.

## Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
int passthru, level, qm_status;
...
qm_status = dpci_qm_get(&passthru, &level);
if (qm_status != 0)
{
    fprintf(stderr, "%s: Can't read quiet mode status: %s\n", argv0,
dpci_last_os_error_string());
    exit(2);
}
qm_status = ((passthru)? 1:0) << 1;
qm_status |= (level)? 1:0;
switch (qm_status)
{
    case 0:
        printf("Quiet Mode Status: ON - Passthrough \n");
        break;
    case 1:
        printf("Quiet Mode Status: OFF - Passthrough\n");
        break;
    case 2:
        printf("Quiet Mode Status: ON - Set\n");
        break;
    case 3:
        printf("Quiet Mode Status: OFF - Set\n");
        break;
    default:
        fprintf(stderr, "%s: can't read quiet mode status: %s\n", argv0,
dpci_last_os_error_string());
        break;
}
```

## 2.16.2 SET QUIET MODE STATUS

### Definitions

```
#include <dpci_core_api.h>

int dpci_qm_set(int passthrough, int level);
```

### Parameters

passthrough	The desired quiet mode passthrough(QMPT) status
level	The desired quiet mode level(QMLV) status

### Description

*dpci\_qm\_set()* allows the programmer to set the status of the QMPT and QMLV bits used to configure quiet mode.

### Return Value

0	The quiet mode status was set.
< 0	An error occurred while reading the quiet mode status

### Notes

Quiet mode is supported only on DPX S- and C-series. For a definition of the parameters, see Figure 59: Quiet Mode Fields and Settings on p250.

### Availability

The *dpci\_qm\_set()* function first appeared in the DirectPCI API version 2.0.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
int passthru, level;
...
if (dpci_qm_set(passthru, level) != 0)
{
    fprintf(stderr,
            "%s: Can't set quiet mode status: %s\n",
            argv0,
            dpci_last_os_error_string());
    exit(2);
}
```

## 2.17 I/O BOARD API

### 2.17.1 CHECK I/O BOARD SUPPORT

#### Definition

```
#include <dpci_core_api.h>

int dpci_io_board_supported(void);
```

#### Parameters

None

#### Description

This function identifies whether the on-board DirectPCI hardware supports an extern I/O board or not, as is possible on ConnectBus-II® systems. Single-board systems such as the DPX-S-series do not support such additional I/O hardware.

#### Return Value

0	I/O board is not supported
1	I/O board is supported
-1	Error occurred during operation.

#### Availability

The *dpci\_io\_board\_supported()* function first appeared in the DirectPCI API version 2.0.0.

#### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

int is_supported = dpci_io_board_supported();
switch (is_supported)
{
case -1:
    fprintf(stderr,
            "I/O board support could not be determined\n");
    exit(-1);

case 0:
    printf("I/O board is not supported on this hardware\n");
    break;

case 1:
    printf("I/O board is supported on this hardware\n");
    break;

default:
}
```

## 2.17.2 GET I/O BOARD IDENTIFIER

### Definition

```
#include <dpci_core_api.h>

int dpci_io_getboard_id(void);
```

### Parameters

None

### Description

This function returns the identification code of any inserted I/O board. If no I/O board is available then 255 (0xff) is returned.

### Return Value

0 to 254	Identification code of inserted board.
255	I/O board is not present
-1	Error occurred during operation.

### Availability

The *dpci\_io\_getboard\_rev()* function first appeared in the DirectPCI API version 1.2.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

#define MY_BOARD_ID 0x03

int board_id = dpci_io_getboard_id();
switch (board_id)
{
case -1:
    fprintf(stderr, "I/O board status could not be determined\n");
    exit(NO_IO_BOARD_ERROR);

case 255:
    fprintf(stderr, "I/O extension board absent\n");
    exit(NO_IO_BOARD_ERROR);

case MY_BOARD_ID:
    break;

default:
    printf("I/O board id is %02x not %02x\n", board_id, MY_BOARD_ID);
    exit(NO_IO_BOARD_ERROR);
}
```

A further example can be found in the *io\_demo.c* program.

## 2.17.3 GET I/O BOARD REVISION

### Definition

```
#include <dpci_core_api.h>

int dpci_io_getboard_rev(void);
```

### Parameters

None

### Description

This function returns the identification code of any inserted I/O board. If no I/O board is available then 255 (0xff) is returned. Depending on the implementation of the I/O board, this might also be a valid revision number. The meaning of the bits in the revision code is determined by the implementer of the I/O board extension.

### Return Value

0 to 255	Revision of inserted board.
-1	Error occurred during operation.

### Availability

The *dpci\_io\_getboard\_rev()* function first appeared in the DirectPCI API version 1.2.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>
#define MIN_BOARD_REV 0x10
...
int board_rev;
board_rev = dpci_io_getboard_rev();
if (board_rev == -1)
{
    fprintf(stderr,
            "I/O board status could not be determined\n");
    exit(BAD_BOARD_REV_ERROR);
}
if (board_rev < MIN_BOARD_REV)
{
    fprintf(stderr,
            "I/O board is rev. %02x but need at least %02x\n",
            board_id, MIN_BOARD_REV);
    exit(NO_IO_BOARD_ERROR);
}
```

A further example can be found in the *io\_demo.c* program.

## 2.17.4 ENABLE I/O BOARD WATCHDOG

### Definition

```
#include <dpci_core_api.h>

int dpci_iowd_enable(void);
```

### Parameters

None

### Description

This function enables the I/O watchdog on I/O boards 80-1003 and 80-0062 on ConnectBus-II® systems. The watchdog has a fixed interval of about 1.7 seconds and must therefore be patted very regularly every second. When the watchdog fires, all digital outputs become tri-state (high-impedance).

### Return Value

0	I/O watchdog was enabled.
-1	Error occurred during operation.

### Notes

This function is not relevant to single board solutions like the DPX-112 and S-series. Hence the API returns an error when used on unsupported hardware.

### Availability

The *dpci\_iowd\_enable()* function first appeared in the DirectPCI API version 2.0.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
if (dpci_iowd_enable() == -1)
{
    fprintf(stderr,
            "I/O board watchdog could not be enabled\n");
}
```

## 2.17.5 DISABLE I/O BOARD WATCHDOG

### Definition

```
#include <dpci_core_api.h>

int dpci_iowd_disable(void);
```

### Parameters

None

### Description

This function disables the I/O watchdog on I/O boards 80-1003 and 80-0062 on ConnectBus-II® systems.

### Return Value

- |    |                                  |
|----|----------------------------------|
| 0  | I/O watchdog was enabled.        |
| -1 | Error occurred during operation. |

### Notes

This function is not relevant to single board solutions like the DPX-112 and S-series. Hence the API returns an error when used on unsupported hardware.

### Availability

The *dpci\_iowd\_enable()* function first appeared in the DirectPCI API version 2.0.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
if (dpci_iowd_disable() == -1)
{
    fprintf(stderr,
            "I/O board watchdog could not be disabled\n");
}
```

## 2.17.6 PAT I/O BOARD WATCHDOG

### Definition

```
#include <dpci_core_api.h>

int dpci_iowd_pat(void);
```

### Parameters

None

### Description

This function pats the I/O watchdog on I/O boards 80-1003 and 80-0062 on ConnectBus-II® systems. The watchdog has a fixed interval of about 1.7 seconds and must therefore be patted regularly.

### Return Value

0	I/O watchdog was enabled.
-1	Error occurred during operation.

### Notes

This function is not relevant to single board solutions like the DPX-112 and S-series. Hence the API returns an error when used on unsupported hardware.

### Availability

The *dpci\_iowd\_enable()* function first appeared in the DirectPCI API version 2.0.0.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
if (dpci_iowd_enable() == -1)
{
    fprintf(stderr,
            "I/O board watchdog could not be enabled\n");
}

while (1)
{
    sleep(1000); // use sleep(1) on Linux
    if (dpci_iowd_pat() == -1)
    {
        exit();
    }
}
```

## 2.18 GPIO API

### 2.18.1 READ GPIO INPUT LINE

#### Definitions

```
#include <dpci_core_api.h>

int dpci_gpio_read_ip(int gpio_line, unsigned char *pvalue);
```

#### Parameters

- gpio\_line      The number of the input line to interrogate.  
 pvalue          Buffer to hold the line status

#### Description

*dpci\_gpio\_read\_ip()* allows the programmer to read the status of one of the GPIO input lines available on some DPX-series mainboards. The status is returned as 0 for 0V or non-zero for >2.2V. The following table lists the GPIO pins available.

Figure 60. List of GPIO Lines

Board	Pin	GPIO Assignment
DPX-S4xx (MUC)	USERIO0	GPIO0
DPX-Cx05 (J18)	USERIO1	GPIO1
DPX-112 (J33)		
DPX-117 (J16)	USERIO2	GPO0
	USERIO3	GPO1
Other boards	None	

Please refer to the user manual for your target DPX-series board for further details of the GPIO configuration.

#### Return Value

- 0            Success – The status bit was read into the *pvalue* buffer  
 < 0          An error occurred while reading the GPIO level

#### Notes

The GPIO lines double-up as OneWire/iButton busses. You should not use a GPIO line for OneWire/iButton purposes while already using it for GPIO purposes.

## **Availability**

The *dpci\_gpio\_read\_ip()* function first appeared in the DirectPCI API version 3.0.0.

## 2.18.2 READ GPIO OUTPUT LINE

### Definitions

```
#include <dpci_core_api.h>

int dpci_gpio_read_op(int gpio_line, unsigned char *pvalue);
```

### Parameters

gpio\_line      The number of the output line to interrogate.

pvalue          Buffer to hold the line status

### Description

*dpci\_gpio\_read\_op()* allows the programmer to read the last state written to one of the GPIO output lines available on some DPX-series mainboards. The status is returned as 0 for 0V or non-zero for >2.2V.

See 2.18.1 *Read GPIO Input Line* for a list of the available GPIO pins.

Please refer to the user manual for your target DPX-series board for further details of the GPIO configuration.

### Return Value

0                Success – The status bit was read into the *pvalue* buffer

< 0             An error occurred while reading the GPIO level

### Notes

The GPIO lines double-up as OneWire/iButton busses. You should not use a GPIO line for OneWire/iButton purposes while already using it for GPIO purposes.

## 2.18.3 WRITE GPIO OUTPUT LINE

### Definitions

```
#include <dpci_core_api.h>

int dpci_gpio_write_op(int gpio_line, unsigned char value);
```

### Parameters

gpio\_line      The number of the output line to write to.

value           The value to write

### Description

*dpci\_gpio\_write\_op()* allows the programmer to set the state of one of the GPIO output lines available on some DPX-series mainboards. The states to set are: 0 for 0V or non-zero for >2.2V.

See 2.18.1 *Read GPIO Input Line* for a list of the available GPIO pins.

Please refer to the user manual for your target DPX-series board for further details of the GPIO configuration.

### Return Value

0               Success – The line state was set

< 0              An error occurred while setting the GPIO level

### Notes

The GPIO lines double-up as OneWire/iButton busses. You should not use a GPIO line for OneWire/iButton

## 2.19 DPX-E130 BACKUP POWER SUPPLY API

The DPX-E130 Backup Power Supply API is available only on the DPX-E130 in order to use the super-cap based system to allow extra power hold up when a power-fail is detected.

On the DPX-E130, digital input 15 is used to detect BPS charging status. Therefore this input may not be used for regular digital IO.

### 2.19.1 ENABLE THE BPS FACILITY

#### Definitions

```
#include <dpci_core_api.h>

int dpci_e130_bps_enable(int enable);
```

#### Parameters

**enable** Boolean value: 0=disable the BPS, 1=enable the BPS.

#### Description

The function `dpci_e130_bps_enable()` disables or enables the BPS, according to whether the parameter *enabled* is zero or non zero respectively.

#### Return Value

0	The BPS was enabled.
< 0	The BPS could not be enabled.

#### Availability

The `dpci_e130_bps_enable()` function first appeared in the DirectPCI API version 3.1.0.

#### Example

See the example in 2.19.3.

## 2.19.2 CHECK BPS ENABLED STATUS

### Definitions

```
#include <dpci_core_api.h>

int dpci_e130_bps_is_enabled(void);
```

### Parameters

None

### Description

The function `dpci_e130_bps_is_charged()` determines whether the back-up power supply is enabled.

### Return Value

>0	The BPS is enabled
0	The BPS is not enabled.
< 0	The BPS status cannot be determined.

### Availability

The `dpci_e130_bps_is_enabled()` function first appeared in version 3.1.0.

### Example

See the example in 2.19.3.

### 2.19.3 CHECK BPS IS CHARGED

#### Definitions

```
#include <dpci_core_api.h>

int dpci_e130_bps_is_charged(void);
```

#### Parameters

None

#### Description

The function `dpci_e130_bps_is_charged()` determines whether the back-up power supply is fully charged.

#### Return Value

- |     |  |
|-----|--|
| >0  | The BPS is charged                     |
| 0   | The BPS is not charged or not enabled. |
| < 0 | The BPS status cannot be determined.   |

#### Availability

The `dpci_e130_bps_is_charged()` function first appeared in version 3.1.0.

#### Example

```
#include <dpci_core_api.h>

int ret;
int timeout_ms = 240000;

ret = dpci_e130_bps_enable(enable);
if (ret == -1)
{
    fprintf(stderr, "Failed to enable/disable BPS: %s.\n",
            dtl_last_os_error_string(ERROR_SUBSYS_OS));
    return;
}

while (timeout_ms > 0)
{
    if (dpci_e130_bps_is_charged())
        break;
    timeout_ms -= 1000;
    sleep(1000);
}
if (timeout_ms > 0)
    fprintf(stderr, "BPS still not charged!\n");
```

## 2.20 MISCELLANEOUS API

### 2.20.1 READ DIP SWITCH STATUS

#### Definition

```
#include <dpci_core_api.h>

int dpci_dis_read(void);
```

#### Parameters

None

#### Description

Reads the state of the motherboard digital DIP switches and returns a bit-mask of switches in the “on” state.

#### Return Value

-1	An error occurred
>=0	A bit-mask is returned with bits corresponding to the state of the switches; bit $n$ in the return value corresponds to SW $n$ e.g. 5 - switches in the state 0101 i.e. switch 1 and 3 are ON. e.g. 8 - switches in the state 1000 i.e. switch 4 is ON.

#### Availability

The *dpci\_dis\_read()* function first appeared in the DirectPCI API version 1.0.0.

#### Example

```
#include <stdio.h>
#include <dpci_core_api.h>

BOOL get_dip_settings(int *dip1, int *dip2, int *dip3, int *dip4)
{
    int ival = dpci_dis_read();
    if (ival == -1)
    {
        return FALSE;
    }
    *dip1 = ival & 0x01;
    *dip2 = ival & 0x02;
    *dip3 = ival & 0x04;
    *dip4 = ival & 0x08;
    return TRUE;
}
```

A further example can be found in the *io\_demo.c* program.

## 2.20.2 WAIT FOR PFD INTERRUPT

### Definitions

```
#include <dpci_core_api.h>

int dpci_io_wait_pfd(unsigned long timeout_ms);
```

### Parameters

`timeout_ms` the maximal time to wait for the PFD event before returning 0.

### Description

`dpci_io_wait_pfd()` blocks the current thread and waits up to `timeout_ms` milliseconds for a power fail detect interrupt to be reported. The function returns if either an interrupt or timeout occurred.

### Return Value

- |     |   |
|-----|---|
| 0   | The function timed-out: no power-fail interrupt occurred. |
| < 0 | An error occurred   |
| > 0 | A PFD interrupt occurred                                  |

### Notes

Power Fail Detect is supported only on DPX S- and C-series.

It is recommended instead to use the new Event Streaming API to detect this event – see page 97.

### Availability

The `dpci_io_wait_pfd()` function first appeared in the DirectPCI API version 2.0.0.

### Example

An example can be found in the `io_demo.c` program.

## 2.20.3 READ SYSTEM BIOS SIZE

### Definitions

```
#include <dpci_core_api.h>

int dpci_bios_size(unsigned long *buff_size);
```

### Parameters

**Buff\_size**      Pointer to the size of buffer passed in “buff”

### Description

*dpci\_bios\_sizer()* allows the programmer to the size of the read-only section of the system BIOS. This value can then be used to allocated memory prior to calling *dpci\_bios\_dump()*.

### Return Value

- |     |  |
|-----|--|
| 0   | The system BIOS memory size was copied into the buffer pointed to by <b>buff_size</b> .  |
| < 0 | An error occurred while reading the BIOS memory size. On Windows, the error code <b>ERROR_NOT_SUPPORTED</b> is returned via <b>GetLastError()</b> if BIOS access is not possible on the host platform. |

### Availability

The *dpci\_bios\_size()* function first appeared in the Direct PCI API version 3.0.0.

### Example

An example usage of this function can be found in the *bios\_demo.c* program.

## 2.20.4 READ SYSTEM BIOS MEMORY

### Definitions

```
#include <dpci_core_api.h>

1xint dpci_bios_dump(byte *buff, unsigned long *buff_size);
```

### Parameters

**Buff**      Buffer to hold the BIOS memory

**Buff\_size**      Pointer to the size of buffer passed in “buff”

### Description

*dpci\_bios\_dump()* allows the programmer to get a snapshot of the system BIOS memory. The BIOS memory refers to the upper 1-4MB of system memory space.

### Return Value

- |     |   |
|-----|---|
| 0   | The system BIOS memory was read into the buffer pointed to by buff.   |
| < 0 | An error occurred while reading the BIOS memory. On Windows, the error code <code>ERROR_NOT_SUPPORTED</code> is returned if BIOS access is not possible on the host platform.<br><br>If on calling <code>dpci_bios_dump()</code> , the variable <code>buff_size</code> points to an integer smaller than the size required to hold the BIOS image, then |

### Notes

The API supports only BIOS memory read. Memory writes are not supported.

### Availability

The `dpci_bios_dump()` function first appeared in the Direct PCI API version 3.0.0.

### Example

An example usage of this function can be found in the `bios_demo.c` program.

## 2.20.5 GET SERIAL IDENTIFIER (DPX-C710 ONLY)

### Definition

```
#include <dpci_core_api.h>

int dpci_core_get_serial(serial_t *serial_id);
```

### Parameters

serial\_id      a pointer to the serial identifier object (The serial\_t structure is defined in dpci\_core\_types.h)

### Description

This function attempts to detect, read and verify a serial identifier from a Dallas Maxim DS28CM00 I<sup>2</sup>C device. If successful, the serial\_id object will be populated and the return code will be set to zero.

### Return Value

0	Success
< 0	An error occurred

### Availability

The *dpci\_core\_get\_serial()* function first appeared in the DirectPCI API version 3.0.3.

### Example

```
#include <stdio.h>
#include <dpci_core_api.h>
...
serial_t device_serial;

if(!dpci_core_get_serial(&device_serial))
{
    int i;

    printf("Device serial: ");
    for(i=0; i<sizeof(device_serial); i++)
    {
        printf("%2x", device_serial.chars[i]);
    }
    putchar('\n');
}
else
{
    puts("No serial identifier available");
}
```

Read and output the serial identifier.

## 2.21 DEBUG API

### 2.21.1 SET\_DPCI\_CORE\_DRIVER\_DEBUG\_LEVEL

#### Definition

```
#include <dpci_core_api.h>

int dpci_core_set_debug_level(unsigned int level);
```

#### Parameters

level            debug level bitmask to set

#### Description

This function sets the debug level parameter for the dpci\_core driver. This determines which debug messages are emitted by the driver. Note that debug messages are emitted only by drivers built for this purpose. Regular release drivers do not emit such messages for reasons of efficiency.

Please see section 1.7 for further information about debug drivers and driver levels.

#### Return Value

>= 0	Success
< 0	An error occurred

#### Availability

The dpci\_core\_set\_debug\_level() function first appeared in the DirectPCI API version 3.0.0.

## 2.21.2 GET DPCI\_CORE DRIVER DEBUG LEVEL

### Definition

```
#include <dpci_core_api.h>

Unsigned int dpci_core_get_debug_level(void);
```

### Parameters

None

### Description

This function returns the current debug level parameter from the dpci\_core driver. This determines which debug messages are emitted by the driver. Please see section 1.7 for further information about debug drivers and driver levels.

#### Return Value

- |     |                     |
|-----|---------------------|
| > 0 | Current debug level |
| < 0 | An error occurred   |

### Availability

The `dpci_core_get_debug_level()` function first appeared in the DirectPCI API version 3.0.0.

## 2.21.3 SET API LIBRARY DEBUG LEVEL

### Definition

```
#include <dpci_core_api.h>

unsigned int dpci_api_set_debug_level(unsigned int level);
```

### Parameters

**level**      The new level to set

### Description

This function sets the bitmask of debug levels used by the DirectPCI library (libdpci.dll or libdpci.so).

This function is only effective in the debug version of the library. In the release version of the library, no debug messages are supported and so this function always returns 0.

Please see section 1.7 for further information about debugging.

### Return Value

**>= 0**      The new resulting debug level

### Availability

The `dpci_api_set_debug_level()` function first appeared in the DirectPCI API version 3.1.1.

## 2.21.4 MODIFY API LIBRARY DEBUG LEVEL

### Definition

```
#include <dpci_core_api.h>

Unsigned int dpci_api_modify_debug_level(unsigned int set,
                                         unsigned int clear,
                                         unsigned int toggle);
```

### Parameters

<b>set</b>	bits to set in the current debug level
<b>clear</b>	bits to clear in the current debug level
<b>toggle</b>	bits to invert in the current debug level

### Description

Manipulate the current debug level by setting, clearing or inverting selected individual bits.

In the release version of the library, no debug levels are supported so this function always returns 0.

Please see section 1.7 for further information about debugging.

### Return Value

>= 0	The new resulting debug level
------	-------------------------------

### Availability

The `dpci_api_modify_debug_level()` function first appeared in the DirectPCI API version 3.1.1.

## 2.21.5 GET API LIBRARY DEBUG LEVEL

### Definition

```
#include <dpci_core_api.h>

Unsigned int dpci_api_get_debug_level(void);
```

### Parameters

None

### Description

This function returns the current API debug level.

In the release version of the library, no debug levels are supported so this function always returns 0.

Please see section 1.7 for further information about debugging.

### Return Value

>= 0	Current debug level
------	---------------------

### Availability

The `dpci_api_get_debug_level()` function first appeared in the DirectPCI API version 3.1.1.

## 2.21.6 SET API LIBRARY DEBUG MESSAGE CALLBACK

### Definition

```
#include <dpci_core_api.h>
#include <dpci_api_types.h>

#define DPCI_DEBUG_CALLBACK_RET_NOLOG 0 /* don't log to file */
#define DPCI_DEBUG_CALLBACK_RET_LOG   1 /* log message to file */

typedef int (*dpci_debug_callback_t)(unsigned int flags, const char
*message);
void dpci_api_set_debug_message_callback(dpci_debug_callback_t func);
```

### Parameters

**func**            The function to call when a message is output.

### Description

This function allows the caller to determine that a specific function must be called whenever a error message or debug message is to be emitted by the DirectPCI Run-time library (libdpci.dll).

Only one such function may be registered in this way. Subsequent calls to this function replace the function pointer provided by previous callers.

Passing a NULL pointer as the *func* parameter disables callbacks when messages are to be emitted.

In the release library, this function performs no action.

When such a message processing callback is in effect, no further output is made to the console window. If the callback is cancelled then output to the console window is resumed.

The *flags* value passed to the callback function represents the severity of the message and the subsystem that generated it. The message pointed to by *message* is fully formatted and needs no further processing. The message buffer memory is owned by the system and should not be modified or freed.

The return value from the user-supplied callback function determines if any further handling of this message should occur: if the return value is DPCI\_DEBUG\_CALLBACK\_RET\_NOLOG then the message will not also be output to any configured log file (see 2.21.7). Alternatively, if the return value is DPCI\_DEBUG\_CALLBACK\_RET\_LOG then the message will also be appended to a configured log file.

It is not possible to request callbacks for debug or error messages issued by the kernel-mode drivers (dpci\_core.sys and dpci\_mem.sys).

Please see section 1.7 for further information about debugging.

## Return Value

None

## Availability

The `dpci_api_set_debug_callback_message()` function first appeared in the DirectPCI API version 3.1.1.

## Example

```
#include <dpci_core_api.h>
#include <dpci_api_types.h>

int my_debug_callback(unsigned int level, const char *message)
{
    int ret = DPCI_DEBUG_CALLBACK_RET_LOG;

    if ((level & DPCI_DEBUG_SEVERITY_MASK) == DPCI_DEBUG_SEVERITY_ERROR)
    {
        my_error_popup(message);
    }
    else if ((level & DPCI_DEBUG_SEVERITY_MASK) <
              DPCI_DEBUG_SEVERITY_INFO)

    {
        ret = DPCI_DEBUG_CALLBACK_RET_NOLOG;
    }
    return ret;
}

void my_init_debug(void)
{
    dpci_api_set_debug_callback_message(my_debug_callback);
}
```

## 2.21.7 SET API LIBRARY DEBUG OUTPUT FILE

### Definition

```
#include <dpci_core_api.h>

int dpci_api_set_debug_output_file(const char *path);
```

### Parameters

path            The file to write debug information

### Description

Set the path of the file to which debugging information will be written. If the first character in the path is a '+' character then the character will be skipped and the output file will be opened so that new messages are appended at the end. Otherwise, the file will be truncated at the beginning so that all previous content is overwritten.

In the release library, this function performs no action: no file is opened.

Note that output to the log file may be filtered if a callback function has been installed (see 2.21.6) and said callback returns DPCI\_DEBUG\_CALLBACK\_RET\_NOLOG.

Please see section 1.7 for further information about debugging.

### Return Value

0	Success
-1	An error occurred

### Availability

The `dpci_api_set_debug_output_file()` function first appeared in the DirectPCI API version 3.1.1.

### Example

```
#include <dpci_core_api.h>

#define MY_LOG_FILE "c:\\windows\\logs\\dpci-debug.log"

void my_debug_init(void)
{
    if (!dpci_api_set_debug_output_file(MY_LOG_FILE))
    {
        fprintf(stderr,
                "Fatal error opening log file %s: %s\n",
                MY_LOG_FILE,
                dpci_last_os_error_string());
        exit(1);
    }
}
```

## 2.21.8 SET API LIBRARY DEBUG OUTPUT HANDLE

### Definition

```
#include <dpci_core_api.h>

#ifndef WIN32
void dpci_api_set_debug_output_handle(Handle hnd);
#endif defined(linux)
void dpci_api_set_debug_output_handle(int fd);
#endif
```

### Parameters

handle, fd     The handle (Windows) or file descriptor (Linux) to which debug information will be written

### Description

Set the file handle to which the DirectPCI API library will write debugging messages. An existing file handle or descriptor will be closed only if it was opened by the DirectPCI library itself because `dpci_set_debug_output_file()` was called or `DPCI_DEBUG_LOGFILE` was set in the environment. A handle passed to this function during a previous call will remain open.

It is not possible to use POSIX file descriptor API with this function under Windows.

Please see section 1.7 for further information about debugging.

In the release library, this function performs no action.

### Return Value

Nothing

### Availability

The `dpci_api_set_debug_output_handle()` function first appeared in the DirectPCI API version 3.1.1.

## 2.22 BOARD IDENTIFICATION API

This internal API has been partially released for customer use. The header file includes more functions than are documented in this section. Customers should avoid undocumented functions as said functions are reserved for internal use and are liable to unpredictable and undocumented change.

### 2.22.1 GET THE BOARD HOST CODE

#### Definition

```
#include <dpci_boards.h>

#define DPX_NONE          0
#define DPX_112            1
#define DPX_116            2
#define DPX_116U           3
#define DPX_117            4
#define DPX_S410           5
#define DPX_S305           6
#define DPX_C705C605        7
#define DPX_E105S          8
#define DPX_E105F          9
#define DPX_E115           10
#define DPX_S415           11
#define DPX_S420           12
#define DPX_S425           13
#define DPX_E120S          14
#define DPX_E120F          15
#define DPX_S430           16
#define DPX_C710           17
#define DPX_S435           19
#define DPX_E130           20

int dpci_board_get_host_board_code(void);
```

#### Parameters

None

#### Description

This function identifies the board on which the software is running. It returns the code for that board. If the board cannot be identified then DPX\_NONE is returned. The identification is implemented by matching the PCI vendor and device IDs of the PCI host bridge or PCIe root complex against a table of recorded values.

- In Linux, these values are obtained by the driver so board identification depends always upon the dpci\_core driver having been loaded successfully. Note however

that the *dpxname* utility parses the output of ‘lspci –n’ and therefore it does not require the dpci\_core driver to be loaded.

- In Windows, these PCI IDs are obtained from the Windows Registry via the Setup API. As such, the dpci\_core driver is not required to be loaded for board identification to be completed as is the case on Linux.

## Return Value

DPX_NONE	The board could not be identified.
> 0	The code for the host board.

## Availability

The `dpci_board_get_host_board_code()` function first appeared in the DirectPCI API version 3.1.1.

## Example

```
#include <dpci_boards.h>

void my_board_id(void)
{
    int board_id;

    board_id = dpci_board_get_host_board_code();
    if (board_id == DPX_NONE)
    {
        fprintf(stderr, "Could not identify host board. Exiting!");
        exit(1);
    }
    printf("Board ID is %d.\n", board_id);
}
```

Further examples can now be found in most of the demonstration programs.

## 2.22.2 GET THE BOARD HOST NAME

### Definition

```
#include <dpci_boards.h>

const char *dpci_board_get_host_board_name(void);
```

### Parameters

None

### Description

This function returns the name for the host board. The name is the regular title for the board as used in titles on all official documentation: main letters are in upper case and a hyphen ('-') character is used; for example, "DPX-S435".

If the host board cannot be identified then a NULL pointer is returned.

The function `dpci_board_get_host_board_name()` is equivalent to calling `dpci_board_get_name_from_code()` and passing it the return code from `dpci_board_get_host_board_id()`.

### Return Value

NULL	The host board is not known.
<code>!= NULL</code>	Pointer to the name of the host board as a NUL-terminated string.

### Availability

The `dpci_board_get_host_board_name()` function first appeared in the DirectPCI API version 3.1.1.

```
#include <dpci_boards.h>

const char *board_name = dpci_board_get_host_board_name();
if (!board_name)
{
    fprintf(stderr, "Could not identify host board. Exiting!");
    exit(1);
}
printf("Board name is %s.\n", board_name);
```

Further examples can now be found in most of the demonstration programs.

## 2.22.3 GET THE BOARD HOST TAG

### Definition

```
#include <dpci_boards.h>

const char *dpci_board_get_host_board_tag(void);
```

### Parameters

None

### Description

This function returns the name for the host board. The tag is a convenient string for the board and is used primarily in developer activities: main letters are in lower case and an underscore ('\_') character is used; for example, "dpx\_s435".

If the host board cannot be identified then a NULL pointer is returned.

The function `dpci_board_get_host_board_tag()` is equivalent to calling `dpci_board_get_tag_from_code()` and passing it the return code from `dpci_board_get_host_board_id()`.

### Return Value

NULL	The host board is not known.
<code>!= NULL</code>	Pointer to the host board tag as a NUL-terminated string.

### Availability

The `dpci_board_get_host_board_tag()` function first appeared in the DirectPCI API version 3.1.1.

### Example

```
#include <dpci_boards.h>

void my_board_id(void)
{
    const char *board_name;

    board_name = dpci_board_get_host_board_tag();
    if (!board_name)
    {
        fprintf(stderr, "Could not identify host board. Exiting!");
        exit(1);
    }
    printf("Board tag is %s.\n", board_name);
}
```

Further examples can now be found in most of the demonstration programs.

## 2.22.4 TRANSLATE A BOARD CODE NUMBER TO BOARD NAME

### Definition

```
#include <dpci_boards.h>

const char *dpci_board_get_name_from_code(int code);
```

### Parameters

code            The board code to translate

### Description

This function returns the board name for a given board code. The board code should be one listed in dpci\_boards.h or one that has been returned by the function dpci\_board\_get\_host\_board\_code(). A description of the form of board names is given in section 2.22.2.

### Return Value

NULL	The board given by the parameter <i>code</i> is not known.
!= NULL	Pointer to the name of the host board name as a NUL-terminated string.

### Availability

The *dpci\_board\_get\_name\_from\_code()* function first appeared in the DirectPCI API version 3.1.1.

### Example

```
#include <dpci_boards.h>

struct my_config
{
    int board_code;
    ...
};

struct my_config *my_config;

void my_check_board(void)
{
    if (my_config->board_code != dpci_board_get_host_board_code())
    {
        fprintf(stderr,
                "ERROR: this game was initialised on a %s but this board
is a %s. Exiting.\n",
                dpci_board_get_name_from_code(my_config->board_code),
                dpci_board_get_host_board_name())
        exit(1);
    }
}
```

## 2.22.5 TRANSLATE A BOARD CODE NUMBER TO BOARD TAG

### Definition

```
#include <dpci_boards.h>

const char *dpci_board_get_tag_from_code(int code);
```

### Parameters

code	The board code to translate
------	-----------------------------

### Description

This function returns the board name for a given board code. The board code should be one listed in dpci\_boards.h or one that has been returned by the function dpci\_board\_get\_host\_board\_code(). A description of the form of board tags is given in section 0.

### Return Value

NULL	The board given by the parameter <i>code</i> is not known.
<i>!= NULL</i>	Pointer to the name of the host board tag as a NUL-terminated string.

### Availability

The *dpci\_board\_get\_name\_from\_code()* function first appeared in the DirectPCI API version 3.1.1.

### Example

```
#include <dpci_boards.h>

struct my_config
{
    int board_code;
    ...
};

struct my_config *my_config;

void my_check_board(void)
{
    if (my_config->board_code != dpci_board_get_host_board_code())
    {
        fprintf(stderr,
                "ERROR: this game was initialised on a %s but this board
is a %s. Exiting.\n",
                dpci_board_get_name_from_code(my_config->board_code),
                dpci_board_get_host_board_name())
        exit(1);
    }
}
```

Page Intentionally Blank

# 3

## WINDOWS EMBEDDED SUPPORT

The DirectPCI SDK provides support for Microsoft Windows XP Embedded, Microsoft Windows Embedded Standard 2009, Microsoft Windows Embedded Standard 7 and Microsoft Windows8 Embedded Standard. The support for the DirectPCI SDK & Run-time incorporates not only components for the DirectPCI drivers and run-time software, but also macro components to help you get started producing embedded Windows XP configurations for your Innocore DPX-series system.

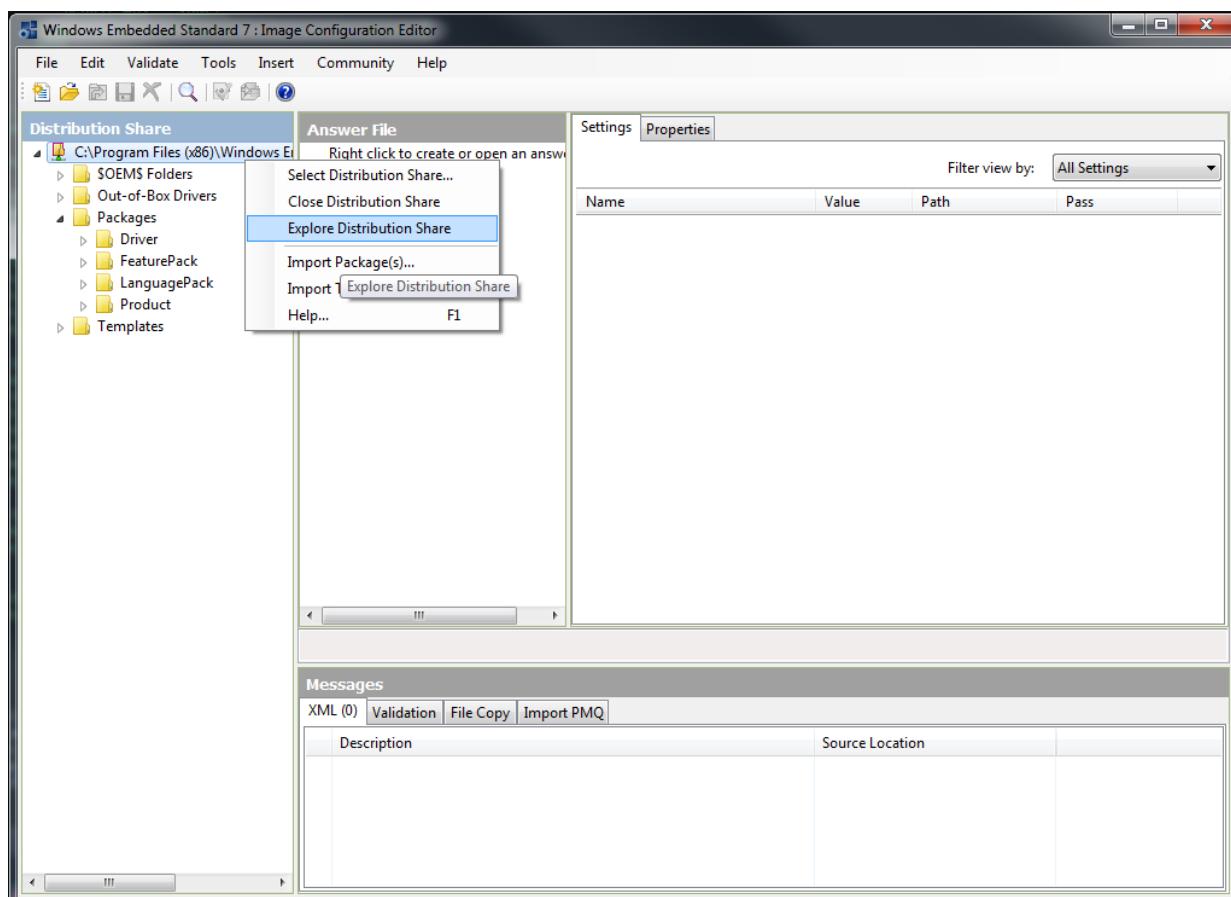
### 3.1 WINDOWS EMBEDDED STANDARD 7, WINDOWS 8 EMBEDDED

The Windows Embedded Standard 7 Image Configuration Editor tool allows you to customise Windows Images. The Image Configuration Editor allows you to create OEM folders where you can add files and folders to a Windows Image, streamlining the installation process.

Any files added to the answer file from the distribution share, the path of the file is included in the answer file. When installing, the Windows Image Builder will use this path to install the file. When you use a distribution share by using Image Configuration tool the folders \$OEM\$ Folders, Out-of-Box Drivers, and Packages are created. You will be able to use \$OEM\$ folders to include logos and branding plus any other application or files needed to complete an installation.

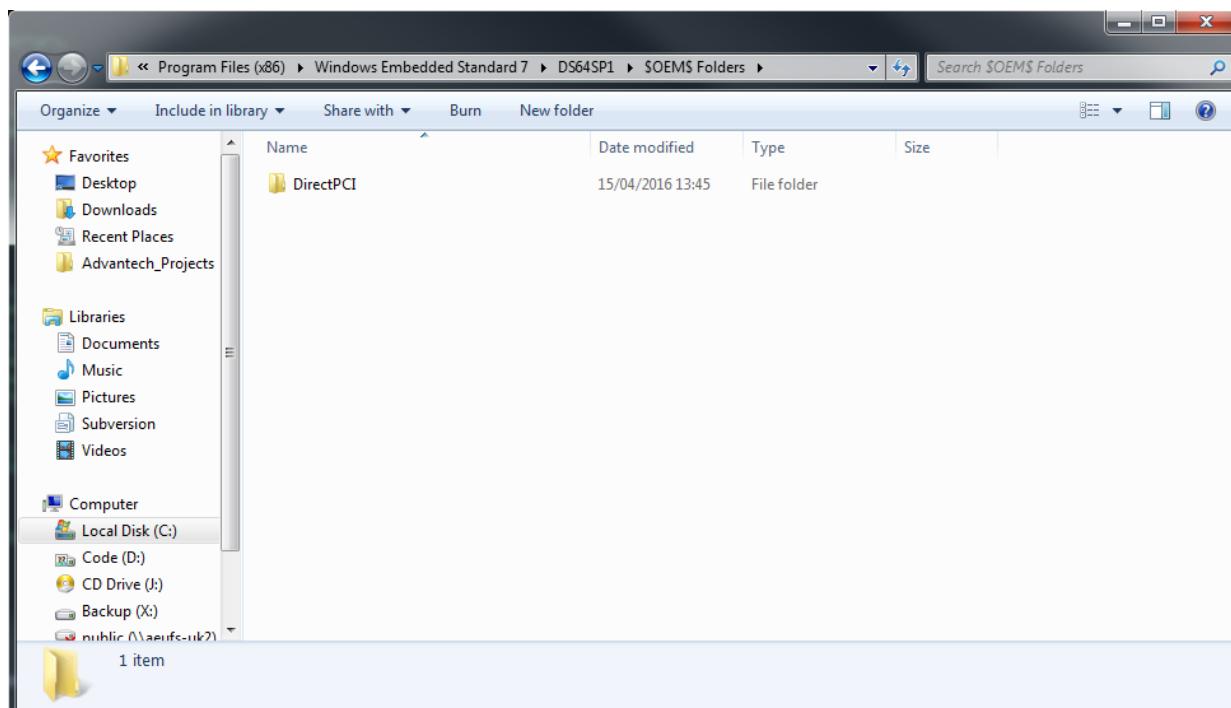
For example, you can add the DirectPCI library needed to run DirectPCI. To add libdpci.dll to your Windows image and into System32 open the Distribution Share pane in ICE, right-click the distribution-share path, and then click Explore Distribution Share.

Figure 61. Distribution Share Menu



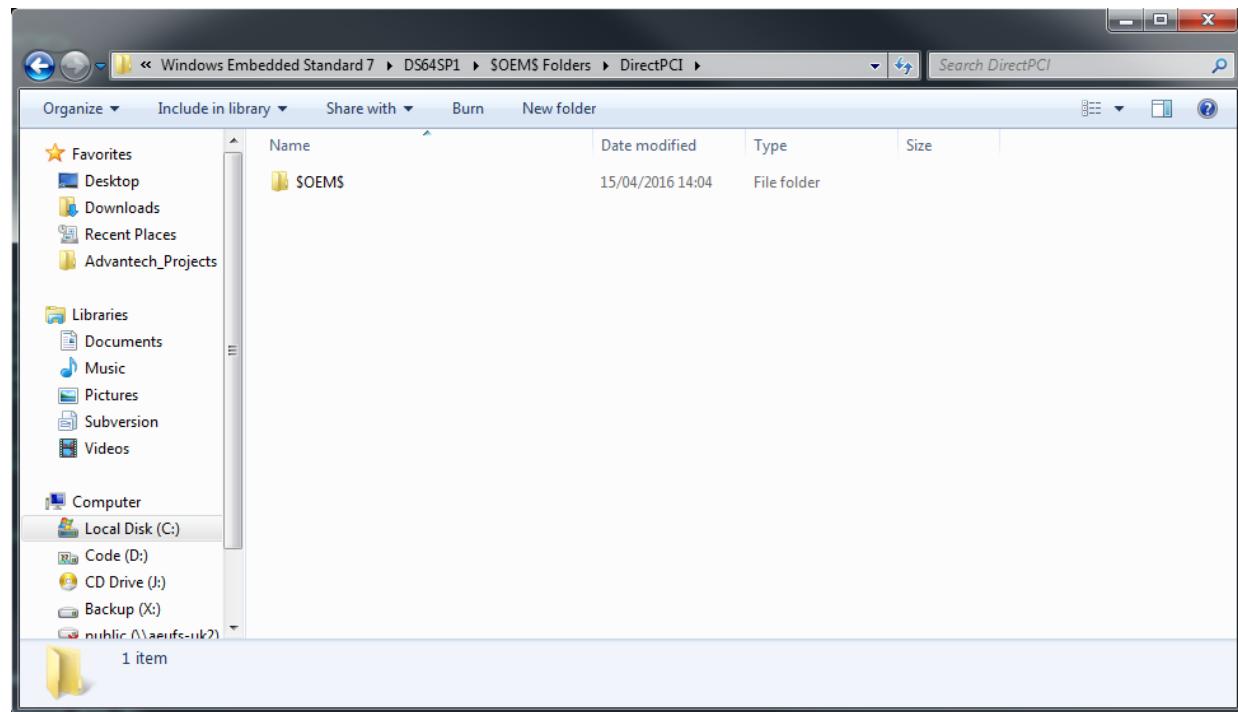
Then Navigate to the \$OEM\$ Folders directory, and then add a new subfolder called "DirectPCI".

Figure 62. DirectPCI folder



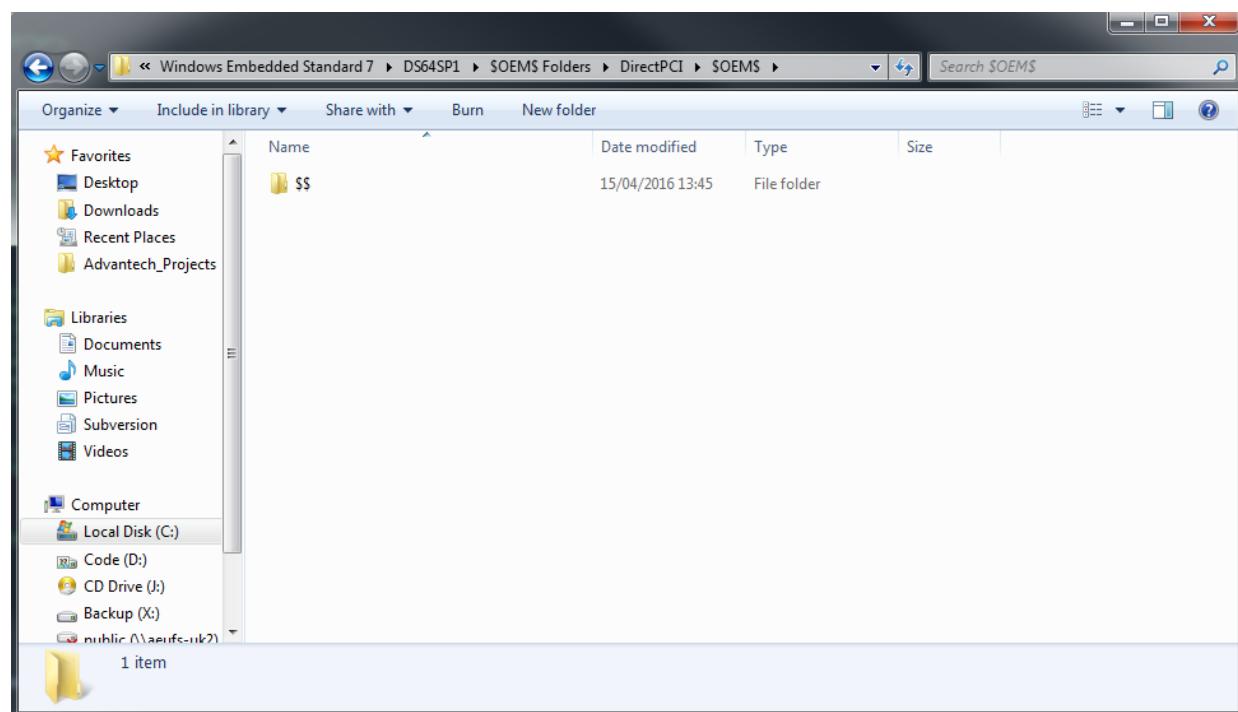
Under “DirectPCI”, add a new subfolder called “\$OEM\$” please note that you must name this subfolder “\$OEM\$”, including the \$ characters, as shown in this example.

Figure 63.     \$OEM\$ Folder



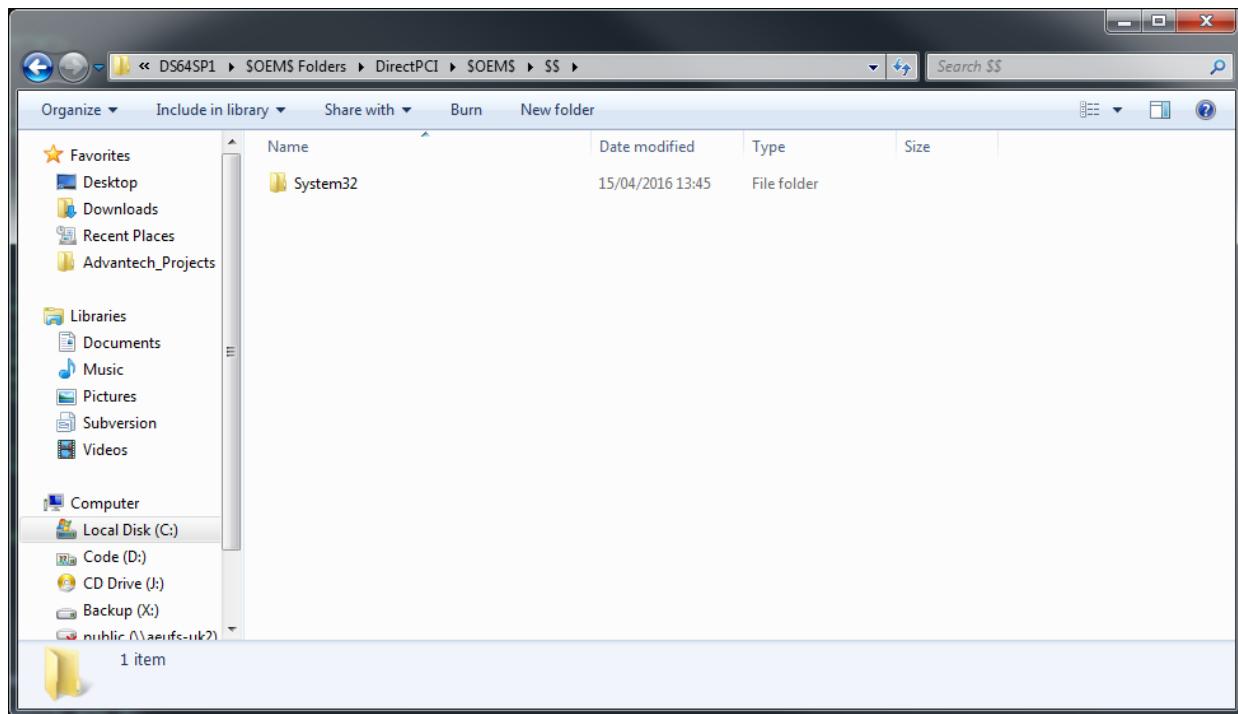
In the “\$OEM\$” create a folder called “\$\$” please note that you must name this subfolder “\$\$”.

Figure 64.     \$\$ Folder



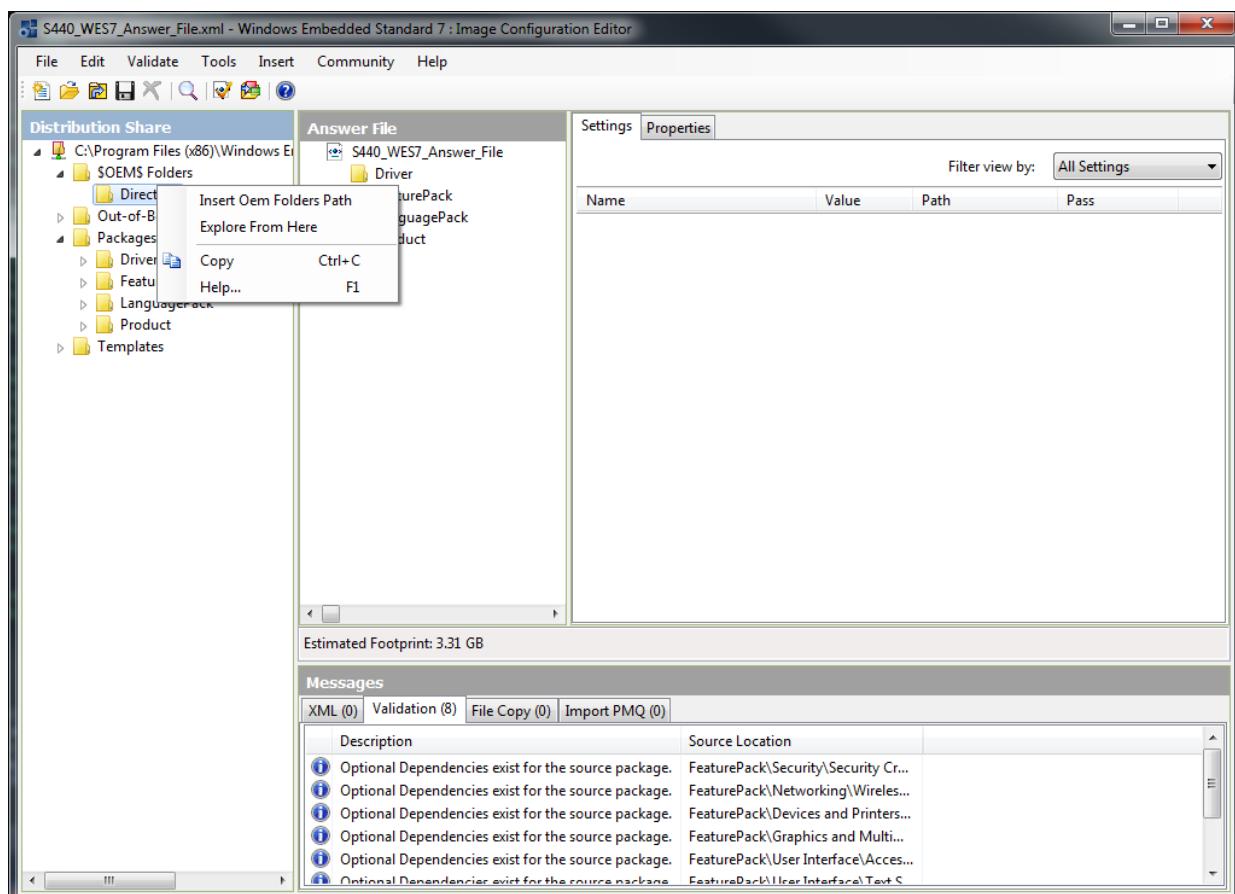
In the “\$\$” create a folder called “System32” and copy the libdpci.dll into it.

Figure 65. System32 folder



To add “\$OEM\$” folder to an answer file, in the Distribution Share in Image Configuration Editor, expand the \$OEM\$ Folders, right click DirectPCI folder and select Insert Oem Folder Path.

Figure 66. Insert Oem Folder Path



You can deploy 32bit applications on a 64bit Windows installation using \$OEM\$ folders however modify the folder structure to use the SysWOW64 folder. Advantech innocore also offer a premade folder structure that you can simply copy available online. [AD: AMAR – EXPLAIN, PLEASE]

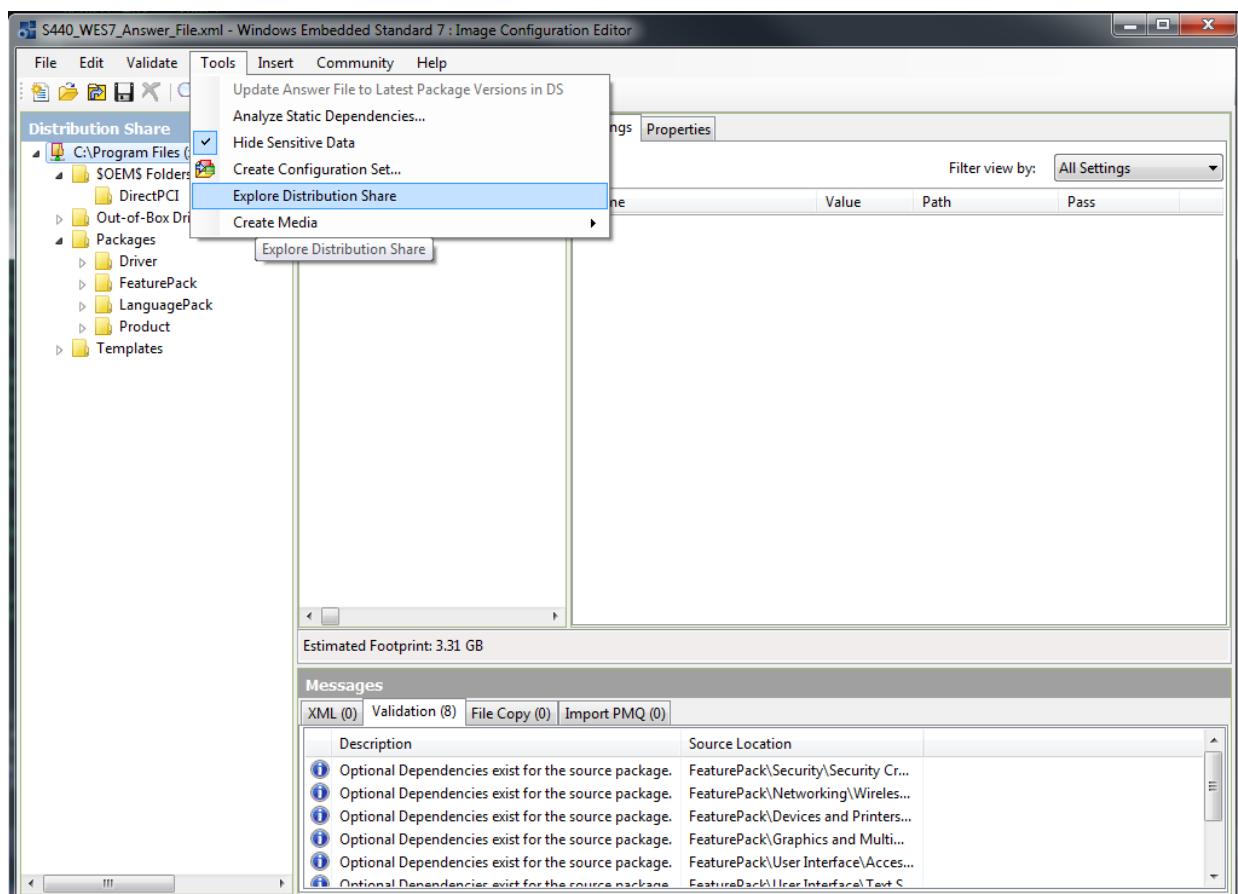
For more details on using \$OEM\$ folders and \$OEM\$ folders structure please look at the following <https://msdn.microsoft.com/en-us/library/ff794890.aspx> and <https://msdn.microsoft.com/en-us/library/ff794652.aspx> MSDN articles.

The Image Configuration Editor also allows Out of box drivers to be added to a Windows Image to make the installation process shorter. Out of box drivers are drivers that are not included in a standard Windows install. Please note that you can copy device drivers directly to the Out of box folder in a distribution share without opening Image Configuration editor.

You can create subfolders in Out of Box drivers to keep some organisation and all drivers in subfolders will be installed during installation.

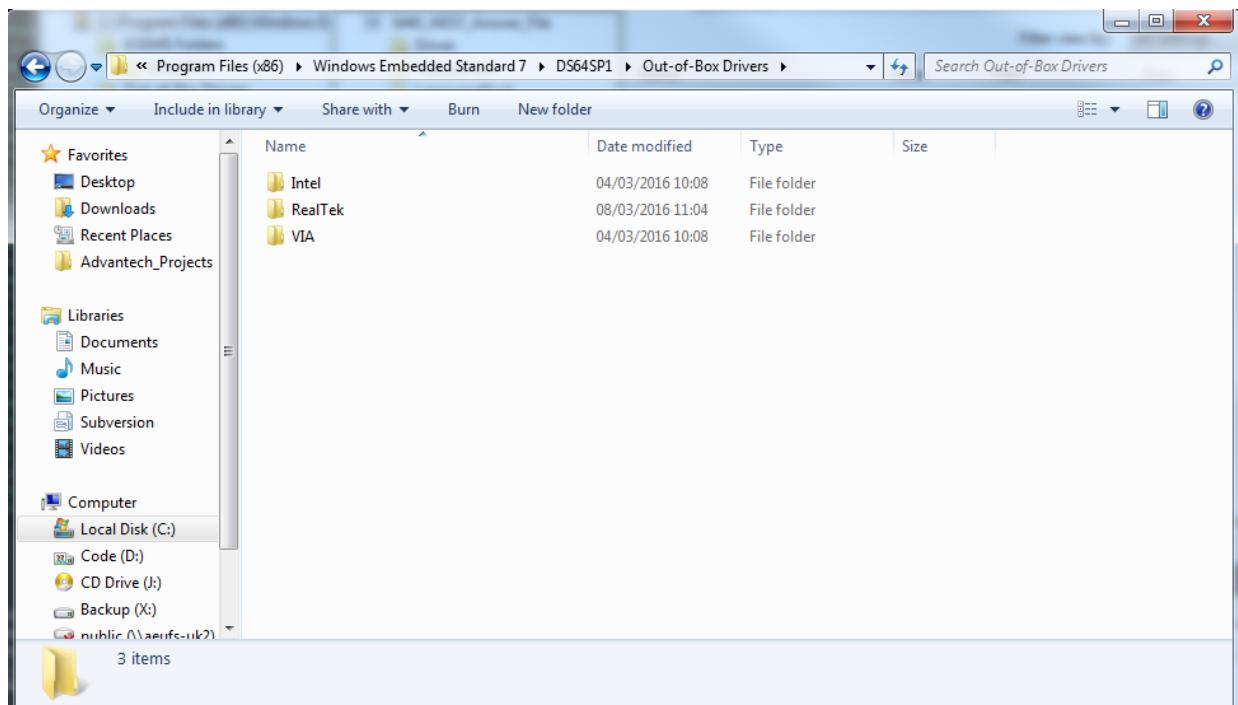
For example to add a device driver, open distribution share an on the Tools menu select Explore Distribution share.

Figure 67. Explore Distribution Share Menu



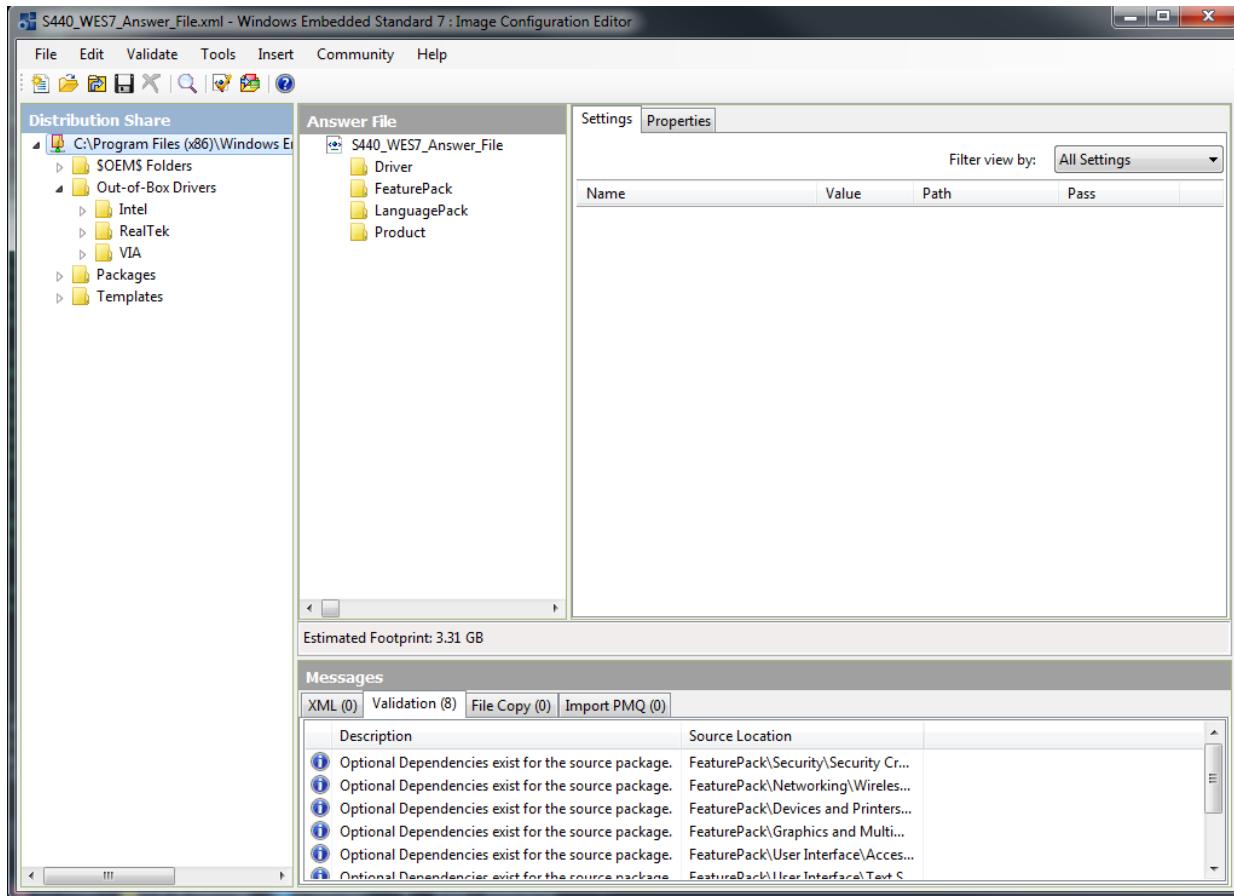
Copy the driver files to Out-Of-Box Drivers folder or you can use drag and drop.

Figure 68. Out-of-Box Drivers Folder



Out of box drivers files should now appear under the out of box drivers node in distribution share pane.

Figure 69. Distribution Share Pane Out-of-box Drivers



You can now use this driver in the image you are creating or any other in the future using the same answer file. If you wish to do this, do not delete the source files of the device drivers or change the folder path without updating the Image Configuration tool with the new location. For more information on Out of box drivers please look at [MSDN article](https://msdn.microsoft.com/en-us/library/ff795041(v=winembedded.60).aspx).

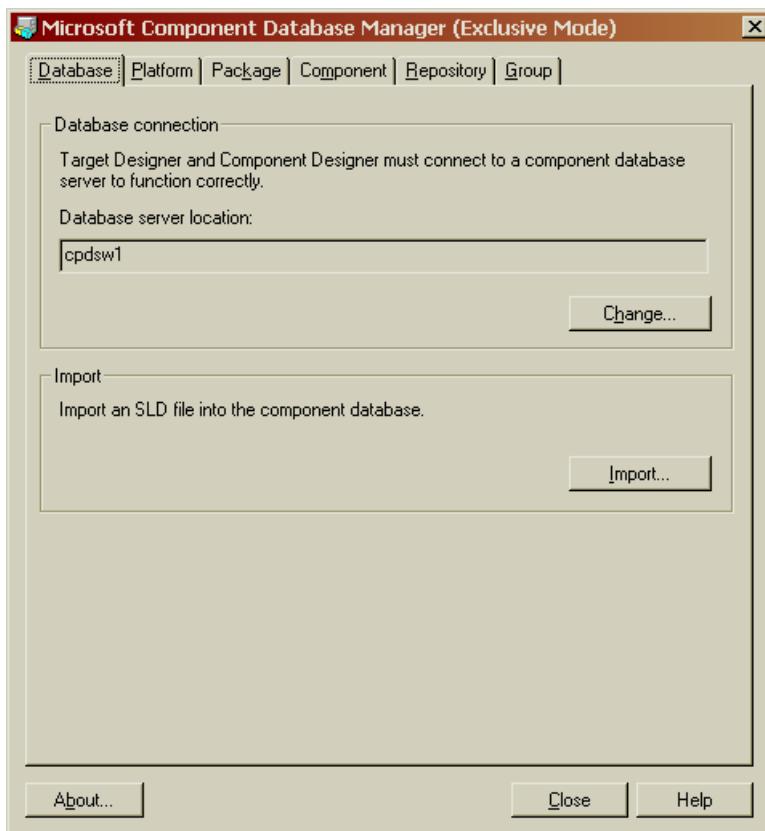
## 3.2 WINDOWS XP EMBEDDED, WINDOWS EMBEDDED STANDARD 2009

### 3.2.1 COMPONENT INSTALLATION

The Windows embedded component files are copied to your developer workstation as part of the DirectPCI SDK installation. However, the components must still be imported manually into your Windows XP Embedded component database before they can be used. It is not necessary to delete previous DirectPCI packages from the database before importing new packages.

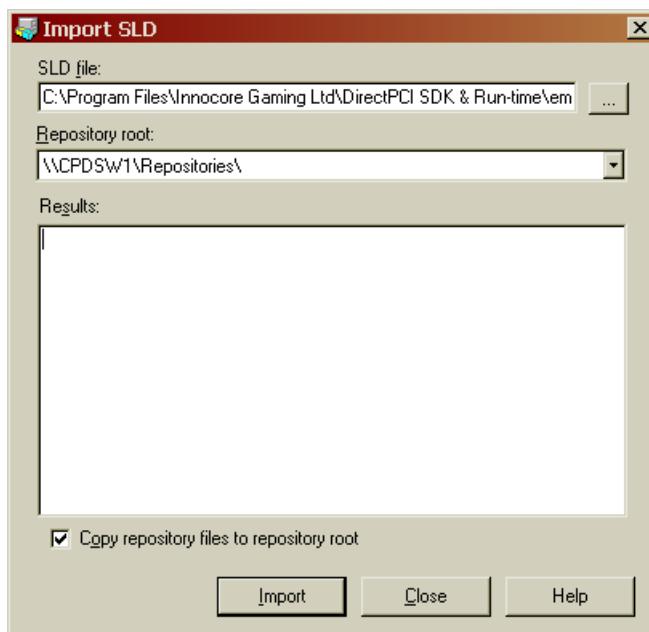
To import the additional components, close any target or component designer applications you have running and start up the Microsoft Component Database Manager in exclusive mode. The Microsoft Component Database Manager can be found under the Microsoft Windows Embedded Studio program group.

Figure 70. Component Database Manager Window



Now hit the Import button and the Import SLD window should appear thus:

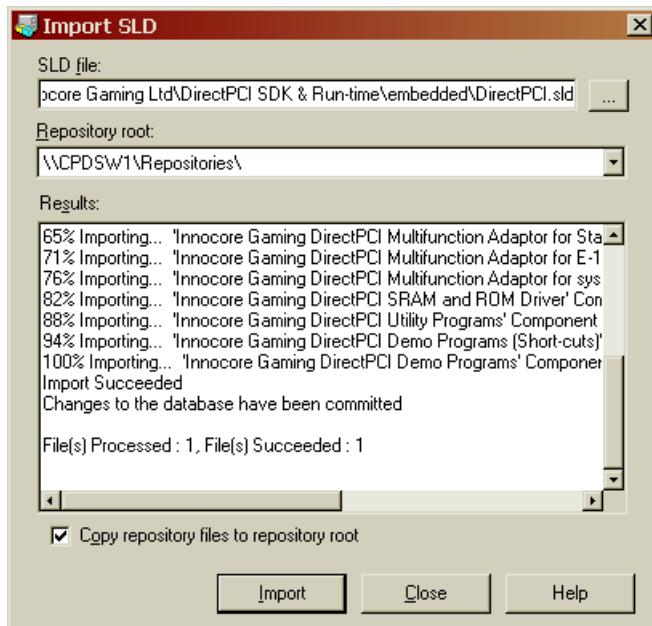
Figure 71. Importing Components into Microsoft Windows XP Embedded



Browse to the DirectPCI.sld file (press ...) which can be found under the *embedded* folder where you installed the DirectPCI SDK. For example, on a regular installation, this file can be found as: C:\Program Files\ Advantech Innocore\DirectPCI SDK & Run-time\embedded\DirectPCI.sld.

Once you have chosen the file, ensure the “Copy repository files to repository root” option is ticked and then press Import:

Figure 72. Output Window on Successful Component Import



The output should resemble the output shown above. You have successfully installed the support for Innocore DirectPCI hardware for Microsoft Windows XP embedded. You may now close the Import SLD and Component Database Manager windows.

### 3.2.2 COMPONENTS

The following components are provided. The dependencies have been carefully verified so that you should be able to add these into existing configurations without worrying about missing files or components.

- Advantech Innocore *DirectPCI Application Run-time*: provides run-time support for applications that need the DirectPCI DLL library.
- Advantech Innocore *DirectPCI Multifunction Adapter for xxxx*: three components provide access to the Digital I/O, watchdog, intrusion detection and serial communications facilities on various different platforms.
- Advantech Innocore *DirectPCI Run-time and Drivers*: combines the application run-time and device driver components in one convenient macro component. You can add this to an existing configuration in order to add support for Innocore DirectPCI hardware.
- Advantech Innocore *DirectPCI SRAM & ROM Drivers*: provides access to the SRAM and ROM facilities.

In addition two utility components are provided to help you test and debug your configurations; these components can be located under the *Software : Test & Development* category.

- Advantech Innocore *DirectPCI Demo Programs*: this component contains binaries compiled from the example code provided with the SDK.

- Advantech Innocore *DirectPCI Utility Programs*: this component contains the utility programs detail in section 4.

More details are provided in the following sections.

### **3.2.3 ADVANTECH INNOCORE DIRECTPCI APPLICATION RUN-TIME**

The *Innocore DirectPCI Application Run-time* support component provides user-land run-time support for applications using the DirectPCI API.

You should include this component in your configuration if you want your application to access the Innocore DirectPCI API using the DLL; you may omit this component if you want your application to use static library instead.

#### **Category Memberships**

This component is located in these categories:

- Software : System : Other

#### **Services**

There are no services associated with this component

#### **Associated Components**

You should also include one or more of the following device driver components in your configuration

- Advantech Innocore DirectPCI Multi-function Controller and I/O Drivers (p.300)
- Advantech Innocore DirectPCI SRAM & ROM Drivers (p.303)

#### **Settings**

There are no configurable settings for this component.

#### **File Resources**

libdpci.dll (to c:\windows\system32)

#### **Registry Resources**

There are no registry resources associated with this component

#### **Dependencies**

There are no dependencies.

#### **Interface Information**

This component exports the DirectPCI Software API.

### **3.2.4 ADVANTECH INNOCORE DIRECTPCI DEMO PROGRAMS**

The *Innocore DirectPCI Demo Programs* component provides a basic facility for testing the installation of DirectPCI run-time and driver components

#### **Category Memberships**

This component is located in these categories:

- *Software : Test & Development*

#### **Services**

There are no services associated with this component.

#### **Associated Components**

You should also include one or more of the following device driver components in your configuration

- *Advantech Innocore DirectPCI Multi-function Controller and I/O Drivers* (p.300)
- *Advantech Innocore DirectPCI SRAM & ROM Drivers* (p.303)

#### **Settings**

There are no configurable settings for this component.

#### **File Resources**

The following files are installed to %PROGRAM\_FILES%\Innocore DirectPCI Demo Programs:

- *bios\_demo.exe*
- *io\_demo.exe*
- *idlp\_demo.exe*
- *idprom\_demo.exe*
- *sram\_demo.exe*
- *rom\_demo.exe*
- *i2c\_demo.exe*
- *eeprom\_demo.exe*
- *ts\_demo.exe*
- *uart\_demo.exe*
- *batt\_demo.exe*
- *callback\_demo.exe*

## Registry Resources

There are no registry resources associated with this component

## Dependencies

This component has a dependency on the following:

- *Advantech Innocore DirectPCI Application Run-time* (p.297)

## Interface Information

This component exports no published or version-controlled interfaces.

### 3.2.5 ADVANTECH INNOCORE DIRECTPCI MULTI-FUNCTION ADAPTOR FOR 80-0062/A I/O BOARD

This component allows you to access the DPX-board's Digital I/O, serial ports, watchdog and intrusion detection facilities from your configuration. This component is tailored for use with supported combinations of a DPX-series main board and a 80-0062/A I/O Board.

#### Category Memberships

This component is located in these categories:

- *Hardware : Devices : Multifunction Adaptors*
- *Hardware : Devices : Ports (COM & LPT)*

#### Services

The *dpci\_core* service is provided by this component.

#### Settings

There are no configurable settings for this component.

#### File Resources

The following files are installed:

- *dpci\_80-0062A.inf* (to c:\windows\inf)
- *dpci\_core.inf* (to c:\windows\inf)
- *dpci\_core.sys* (to c:\windows\system32\drivers)

#### Registry Resources

There are no registry resources associated with this component; however registry resources are installed via the first-boot agent (FBA) activities.

#### Dependencies

This component has the following dependencies:

- Class Installer – Ports (COM & LPT)*  
*Class Installer – Multifunction Adaptors*  
*Communications Port*  
*Generic Multifunction PC-card*

#### Interface Information

This component exports only internal interfaces.

### 3.2.6 ADVANTECH INNOCORE DIRECTPCI MULTI-FUNCTION ADAPTOR FOR DPX-116U & DPX112

This component allows you to access the DPX-board's Digital I/O, serial ports, watchdog and intrusion detection facilities from your configuration. This component is tailored for use with the DPX-116U and DPX-112 mainboard which have on-board I/O.

#### Category Memberships

This component is located in these categories:

- *Hardware : Devices : Multifunction Adaptors*
- *Hardware : Devices : Ports (COM & LPT)*

#### Services

The *dpci\_core* service is provided by this component.

#### Settings

There are no configurable settings for this component.

#### File Resources

The following files are installed:

- *dpci\_multi.inf* (to c:\windows\inf)
- *dpci\_core.inf* (to c:\windows\inf)
- *dpci\_core.sys* (to c:\windows\system32\drivers)

#### Registry Resources

There are no registry resources associated with this component; however registry resources are installed via the first-boot agent (FBA) activities.

#### Dependencies

This component has the following dependencies:

- Class Installer – Ports (COM & LPT)*  
*Class Installer – Multifunction Adaptors*  
*Communications Port*  
*Generic Multifunction PC-card*

#### Interface Information

This component exports only internal interfaces.

### 3.2.7 ADVANTECH INNOCORE DIRECTPCI RUN-TIME AND DRIVERS

The *Innocore DirectPCI Run-time and Drivers* component is a macro component which includes the following components: Run-time, digital I/O and SRAM/ROM driver components.

#### Category Memberships

This component is located in these categories:

- *Software : System : Other*

#### Services

The *dpci\_core* and *dpci\_mem* services are provided by this component.

#### Associated Components

No other components interact with this component.

#### Settings

There are no configurable settings for this component.

#### File Resources

Files to be installed are listed until the File Resources sections of the dependency components.

#### Registry Resources

There are no registry resources associated with this component; however registry resources are installed via the first-boot agent (FBA) activities.

#### Dependencies

This component has the following dependencies:

- Advantech Innocore *DirectPCI Application Run-time* (p.297)
- Advantech Innocore *DirectPCI Multifunction Digital I/O and Serial Drivers* (p.300)
- Advantech Innocore *DirectPCI SRAM and ROM Drivers* (p.303)

#### Interface Information

This component exports only internal interfaces.

### 3.2.8 ADVANTECH INNOCORE DIRECTPCI SRAM AND ROM DRIVERS

The *Innocore DirectPCI SRAM & ROM Drivers* component allows you to access the DPX-board's SRAM and ROM facilities from your configuration.

#### Category Memberships

This component is located in these categories:

- *Hardware : Devices : PCMCIA and Flash memory devices*

#### Services

The *dpci\_mem* service is provided by this component.

#### Associated Components

You should also include in your configuration the *Innocore DirectPCI Application Run-time* component

Do not include the *Hardware : Devices : System Devices: PCI Standard RAM Controller* device as it conflicts with the SRAM and ROM drivers component.

#### Settings

There are no configurable settings for this component.

#### File Resources

The following files are installed:

- *dpci\_mem.inf* (to c:\windows\inf)
- *dpci\_mem.sys* (to c:\windows\system32\drivers)
- *dpci\_rom.inf* (to c:\windows\inf)
- *dpci\_rom.sys* (to c:\windows\system32\drivers)

#### Registry Resources

There are no registry resources associated with this component; however registry resources are installed via the first-boot agent (FBA) activities.

#### Dependencies

This component has the following dependencies:

*Class Installer – PCMCIA and Flash Memory Devices.*

#### Interface Information

This component exports only internal interfaces.

### 3.2.9 ADVANTECH INNOCORE DIRECTPCI UTILITY PROGRAMS

The *Innocore DirectPCI Demo Programs* component provides a basic facility for testing the installation of DirectPCI run-time and driver components

#### Category Memberships

This component is located in these categories:

- *Software : Test & Development*

#### Services

There are no services associated with this component

#### Associated Components

The run-time and digital I/O drivers are associated with this component.

#### Settings

There are no configurable settings for this component.

#### File Resources

The following files are installed to %PROGRAM\_FILES%\Innocore DirectPCI Programs:

- *batt.exe*
- *dipsw.exe*
- *ioboard.exe*
- *idutil.exe*
- *memutil.exe*

#### Registry Resources

There are no registry resources associated with this component

#### Dependencies

This component has dependencies on the following:

- Advantech Innocore DirectPCI Application Run-time (p.297)
- Advantech Innocore DirectPCI Multi-function Controller and I/O Drivers (p.300)

#### Interface Information

This component exports no published or version-controlled interfaces.

# 4

## COMMAND LINE UTILITIES

The DirectPCI SDK includes several command line utilities which can be used in scripts or for general use. They are installed in `/usr/bin` on Linux distributions and the installation bin directory on Windows XP installations. The operation is the same under each operating system.

### 4.1 BATT – SYSTEM BATTERY STATUS

#### Synopsis

```
batt
batt -h
batt -a
batt -v
batt name ...
batt -e
batt -e NAME
batt -e NUM
batt -e name/num mV
batt -c n/m/h/d/o
```

#### Description

The *batt* command allows command-line access to the status of the SRAM back-up batteries. When invoked without any arguments, *batt* gives the status of any batteries that have error conditions.

```
$ batt
```

```
BAT1 FAIL: 1.980V  
$
```

When invoked with the name of a battery, then only that battery's status is displayed.

```
$ batt bat1  
BAT1 OK: 2.980V  
$
```

Starting with the DPX-S series and continuing with the DPX-E and DPX-C series, the batt command also displays the voltage of the battery.

```
$ batt -a  
BAT1 OK: 2.980V  
BAT2 OK: 2.980V  
$
```

Command line options are:

- a Show the status of all batteries, whether in FAIL state or not.
- c Set a time period for automatic battery check. The check period could be once per *min*, *hour*, *day* or *month*; automatic battery check may be disabled by setting the check period to *none*
- e Get and Set the error voltage level for a battery at or below which the battery would be considered invalid.

```
$ batt -e  
BAT1 Error Level: 2.502V  
BAT2 Error Level: 2.502V  
$
```

- h Display a usage message
- v Display the version of the *batt* command.

## Notes

On systems with IDLP firmware v60 or later, reading the battery voltage automatically updates the OK/failure status. This dynamic update does not happen on earlier firmware revisions so battery status information will only be updated according to the regular schedule.

## Availability

Windows XP and Linux: from DirectPCI SDK and Run-time release 1.2.1.

## See Also

[2.14.2 - dpci\\_bat\\_read\(\)](#)

## 4.2 DIPSW – DIP SWITCH STATUS

### Synopsis

```
dipsw  
dipsw -h  
dipsw -v
```

### Description

The *dipsw* command returns the status of the four user dip-switches on the motherboard (where present). The result is a number between 0 and 15 which is the bit-wise combination of all the four bits. This is returned both on the standard output and via the return code.

```
$ dipsw  
0  
$
```

If the user does not have permission to access the DirectPCI device driver or there is a problem with the device or the installation, then 255 is returned.

Command line options are:

- h              Display a usage message
- v              Display the version of the *dipsw* command.

### Availability

Windows XP and Linux: from DirectPCI SDK and Run-time release 1.2.1.

### See Also

[2.20.1 - \*dpci\\_dis\\_read\(\)\* p266](#)

## 4.3 DPCI – GENERAL DIRECTPCI UTILITY

### Synopsis

```
dpci
dpci [-c | -C | -r ]
dpci -D[core|mem]
dpci -D[core|mem] debug-level
dpci -E[adtpi]
dpci -i
dpci -I[dep]
dpci -n
dpci -p input-port
dpci -P output-port value
dpci -q
dpci -q state
dpci -u
dpci -h
dpci -v
```

### Description

The *dpci* command provides the user with a method to identify various system configuration information.

If the user does not have permission to access the DirectPCI device driver or there is a problem with the device or the installation, then 255 is returned.

Command line options are:

- b Show system information.
- c Show the system counters.
- C Show and then reset the system counters.
- D[core|mem] Shows the current debug level of the *dpci\_core* driver.
- D[core|mem] [+|-]*debug-level* Updates the debug word configuration on systems where the *dpci\_core* driver has been built with debugging enabled.  
 The *debug-level* parameter is a number in decimal (or hexadecimal with a leading '0x') which is a bit-mask as defined further below.  
 A leading '+' may be supplied, in which case the supplied number is logical-OR'd with the existing debug-level. If a leading '-' is supplied then the existing debug-level has those bits turned off that are set in the supplied number.

-E[adtpi]	Monitor events as they happen. If no additional letters are supplied then all sources of events are monitored: digital input changes, IDLP events, temperature sensor events and power-fail events. Otherwise, event sources are only tapped as needed. The designations for event sources are thus: a=all, d=digital inputs, t=80-0062 thermal sensor, p=PFD input, i=IDLP events.
-i	Show all the I <sup>2</sup> C ports available.
-Id	Disable the I/O watchdog (80-1003 and 80-0062 I/O boards only)
-le	Enable the I/O watchdog (80-1003 and 80-0062 I/O boards only)
-lp	Pat the I/O watchdog (80-1003 and 80-0062 I/O boards only)
-n	Show the name of the board, for example: DPX-S410, DPX-E115 etc.
-p <i>n</i>	Read the input from the input port <i>n</i>
-P <i>n</i> <i>data</i>	Write <i>data</i> to output port <i>n</i>
-q	Show the quiet mode configuration status
-q <i>status</i>	Configure quiet mode status as <i>status</i> – <i>on</i> , <i>off</i> , <i>pass</i>
-r	Reset the system counters.
-u	Show the 64-bit unique IDPROM ID.
-h	Display a usage message
-v	Display the version of the <i>dpci</i> command.

The *-b* option displays a number of pieces of system information. For example on a DPX-E135 board:

```
# dpci -b
DirectPCI board name:      DPX-E135
DirectPCI board code:       21
DirectPCI firmware ID:     0180
DirectPCI firmware version: 11
IDLP firmware version:     61.6 (0.192)
Power uC supported:        yes
I2C buses:                 1
I/O Board type:            Not supported on standalone boards
Input ports:                4 (4 with interrupts)
Output ports:               4
#
```

Note that some of the field titles have been altered for accuracy in the current maintenance release.

The defined debug levels are shown in Figure 14: Debugging Word Bit-field Values on p40.

## **Availability**

Windows XP and Linux: from DirectPCI SDK and Run-time release 1.4.0.

## 4.4 DPXNAME – SHOW THE NAME OF THE TARGET HARDWARE

### Synopsis

```
dpxname  
dpxname -b  
dpxname -g  
dpxname -h  
dpxname -v
```

### Description

The *dpxname* command is available to allow users to identify which board is in use. The Linux kernel must be configured to support the /proc/pci file in order for *dpxname* to operate correctly.

Invoked without arguments the *dpxname* command displays the name of the DPX-series board and, on Linux, the type of the graphics processor (GPU) installed.

Command line options are:

- b              Display only the type of the DPX-series board
- g              Display only the GPU found. [Linux only]
- h              Display a usage message
- v              Display the version of the *dpxname* command.

### Notes

This command may be removed in the next main release.

On Linux, *dpxname* is a script; on Microsoft Windows systems it is a binary executable.

### Availability

Linux: from DirectPCI SDK and Run-time release 1.2.1.

Windows: from DirectPCI SDK and Run-time release 3.0.0

## 4.5 IDUTIL – IDLP OPERATIONS

### Synopsis

```
idutil -c
idutil -d
idutil -d sync
idutil -d date
idutil -e
idutil -f
idutil -i
idutil -l
idutil -L
idutil -s milliseconds
idutil -t
idutil -w seconds
idutil -W
idutil -w seconds
idutil -v
```

### Description

The *idutil* command allows convenient command-line access to the Intrusion Detection and Logging Processor (IDLP) and the things it monitors.

Command line options are:

- c              Show the IDLP's firmware checksum. This is the checksum of the program code in the chip. This can be used to establish that the code has not changed. The checksum is a simple 16-bit additive checksum.
- d              Show the time and date as maintained by the intrusion detection and logging processor.
- d sync        Set the time and date maintained by the intrusion detection and logging processor to be the system time. The system time used is the adjusted for DST and time-zone dependence. (See the ANSI C *localtime()* function description.)
- d *date*       Sets the time and date maintained by the intrusion detection and logging processor to be the system time. The format of date is YYYYMMDDhhmmss. For example, the date 3<sup>rd</sup> May, 2001 and time 12:34pm and 56 seconds would be encoded as 20010503123456.
- e              Return the number of unread events logged by the intrusion detection and logging processor.
- f              Return the version of firmware running on the intrusion detection and logging processor.

- i      Return the current status of the intrusion inputs monitored by the intrusion detection and logging processor. This facility is supported only on boards where the firmware revision is 24 or newer.
- l      List all unread events kept by the intrusion detection and logging processor.
- L      List previously-read events kept by the IDLP. Events currently queued for reading are not listed. Up to 255 (minus the number of unread events) events may be listed.
- s *milliseconds*      Set the time the intrusion detection and logging processor waits between subsequent checks of the intrusion circuits. The parameter *milliseconds* may have the values 1000, 500, 250 or 125.
- t      Displays a table listing events and the last time the IDLP recorded an instance of that event. Please read first the description of `dpci_id_readevent_instance()` (see 0) for an explanation of limitiations regarding this option.
- w *seconds*      Set the watch-dog time-out period to *seconds* (measured in seconds). The allowable range is 1 – 255 seconds. A setting of zero will disable the watch-dog timer.
- W      Pat (reset) the watch-dog timer.
- W *secs*      Enable the watch-dog and automatically reset its counter. The time-out period is set to *secs* seconds but the watch-dog is patted every *secs* / 2 seconds. The `-W secs` option is available only under Linux.  
  
When invoked with this option, the *idutil* command starts off an independent process (a unix *daemon*) which performs a reset of the IDLP watch-dog every *secs* seconds; this will stop the system being reset automatically.  
  
It is recommended this option only be used during product development and testing. In deployment to customer sites, this option should not be used; instead, application-specific code should be developed to reset the I/O watch-dog. Any errors encountered in normal running are logged to the system log (facility `LOG_DAEMON`, severity `LOG_INFO`) and cause the *daemon* to exit.
- h      Display a usage message
- v      Display the version of the *idutil* command.

## Availability

All platforms from DirectPCI SDK and Run-time release 1.2.1.

**See Also**

[2.8 - DirectPCI IDLP API](#)

## 4.6 IOBOARD - I/O BOARD OPERATIONS

### Synopsis

```
ioboard
ioboard -i
ioboard -r
ioboard -h
ioboard -t
ioboard -T [secs]
ioboard -w
ioboard -v
```

### Description

The *ioboard* command allows command-line access to the type and revision of I/O extension board inserted.

Command line options are:

- i Show the ID of the I/O board inserted.
- r Show the revision of the I/O board inserted.
- W On Linux, automatically refresh the I/O watchdog every second.
- t Show the readings of all known temperature sensors.
- T [secs] Repeatedly show the readings of all known temperature sensors at an interval of secs seconds, or every second if secs is not supplied
- h Display a usage message
- v Display the version of the *ioboard* command.

When invoked without any arguments or with the *-i* argument, *ioboard* gives the ID code of the inserted I/O board, as can be obtained from register I/O 0x10 using the *dpci\_io\_getboard\_id()*. When invoked with the *-r* argument then the revision of the I/O board is displayed, as can be obtained from I/O register 0x11 using the *dpci\_io\_getboard\_rev()*.

Supposing a standard 80-0063 backplane had a model 80-0062, rev.1.1 I/O board inserted, then the following would be produced.

```
$ ioboard -i
21
$ ioboard -r
2
$
```

The *-t* and *-T* options list all known temperature sensors (and their current temperature readings) supported by the DirectPCI facility. Presently these are only available when an 80-0062/A I/O Board II is used in conjunction with a DPX-117 or DPX-C series system.

```
# ./ioboard -t
onboard    remote
      24        28
#
```

If a particular sensor is not connected (on the 80-0062/A board the remote sensor can be disconnected from J4) then the temperature is shown as N/C.

The **-W** option is available only under Linux. When invoked with this option, the *ioboard* command starts off an independent process (a unix *daemon*) which performs a read from digital input port 0 every second. This will stop the digital output ports being disabled by the I/O watch-dog. It is recommended this option only be used during product development and testing. In deployment to customer sites, this option should not be used; instead, application-specific code should be developed to reset the I/O watch-dog. Any errors encountered in normal running are logged to the system log (facility LOG\_DAEMON, severity LOG\_INFO) and cause the *daemon* to exit.

## Availability

Windows XP and Linux: from DirectPCI SDK and Run-time release 1.2.1.

## See Also

*dpci\_io\_getboard\_id()* p254, *dpci\_io\_getboard\_rev()* p255.

## 4.7 MEMUTIL – EEPROM, SRAM, ROM MEMORY ACCESS

### Synopsis

```
memutil -l [-s | -d]
memutil -r -u device [options ...]
memutil -x -u device [options ...]
memutil -w -u device [options ...]
memutil -v
```

### Description

The *memutil* command allows command-line access to the SRAM, ROM and EEPROM facilities provided by DirectPCI on DPX-series main-boards.

When invoked without any arguments, *memutil* displays a terse message indicating the expected usage of the command.

Exactly one of the following key command line options must be used:

- l List the devices which can be accessed. This lists the devices this command can access. With the *-s* flag only the names are listed. Ordinarily the each device's name, access specification (see the *-u* option below), size (if known) and write-capability are listed. With the *-d* flag additional information is provided describing the device and any further information about what the device can achieve.
- r Read from a specified device. The device's memory is accessed and copied to a file specified in subsequent arguments or the *standard output* stream.
- w Write to a specified device. The device's memory is written either from a file specified in subsequent arguments or from the *standard input* stream.
- x Dump contents of a specified device like the Linux *hexdump* command.
- v Show version of the *memutil* command.

The following additional options may be supplied to augment the above options:

- b ... Define the area of memory on the selected device on which the read, write or dump operations will apply.

The additional specifier ... has three forms:

- *start* – access all memory from *start* through the end of the device's memory.
- *start:end* – access the range *start* through *end - 1* inclusive.
- *start+len* – access *len* bytes from the memory at *start*.

If the device selected with *-u* below cannot supply a size for the device

then once of the last two forms above must be used.

The default starting address is always 0; the default ending address is always the end of the device.

- d              Display more information about devices (-/ flag) or operations.
- f ...          Specify a file to receive data read from the device (with -r option) or to supply new data to write to the device (with -w option).  
                 When writing to a device, only as many bytes are written to the device as are available in the file ... and as may be stored in the device; it is not an error if the file size and memory range specified do not match.
- s              Display less information about devices (with -/ flag) – only the names of available devices are listed – one per line.
- u *device*    Specify the device to access. The additional argument *device* specifies the name of the device and any additional data to help identify the device to be used.

The following devices are available (dependent upon the specific platform):

- sram            The static, battery-backed ROM on DPX-series mainboards.
- rom             The ROM on DPX-series mainboards.
- mbee           The EEPROM on DPX-series mainboards since the DPX-116
- ioee           The EEPROM on I/O Board II (80-0062) boards.
- i2c            Any I2C device supported by the system.
- idprom        The One-wire idprom device on new DPX-S and DPX-C series motherboards.
- bios           The system BIOS as accessible in the top portion of the memory map. This is the ROM image from the flash chip and NOT the running BIOS image which is run from RAM. Note also that the memutil cannot be used to write new data to the BIOS flash memory chips.

Figure 73. Devices supported by the memutil command

The -u option can be followed with any of the device Ids above to choose that device.

When choosing an I<sup>2</sup>C device, the -u option must further be supplied with the bus name or number, the device address (in hex with a 0 in the r/w bit position) and the address size in bits. For examples, see the *Examples* section below.

With exceptionally careful use, the i2c access mode can be used to access I<sup>2</sup>C devices other than EEPROMs.

## Availability

Windows XP and Linux: from DirectPCI SDK and Run-time release 1.4.0.

## Examples

The following sections show real usage examples of the *memutil* utility. Lines beginning with a hash-mark (#) are comments in the Unix™ shell-script style.

```
# Dump the entire ROM contents
#
#memutil -x -u ROM

# Write a string into SRAM at location 4096 onward.
#
echo "Hello, world!" | memutil -w -u sram -b 0x1000

# Do this same. This version only copies the first five bytes ("Howdy")
# to address 4096.
#
echo "Howdy, world!" | memutil -w -u sram -b 4096+5

# Get contents of AT24c64 eeprom connected to IO Board II via the
# backplane and put in eeprom.dat. The eeprom address bits are 1010010.
# The I2C R/W bit needs to be inplace and clear hence 1010010 becomes
# 10100100 (0xa4/164)
#
memutil -r -f eeprom.dat -b 0:8192 -u i2c:3:a4
```

The following shows the console output after invoking *memutil -l -d* on a DPX-C705 with an 80-0062 I/O board on a system running GNU/Linux. (Output on Windows system is very similar.)

Name	Spec	Size	Writable
sram	sram On-board battery-backed static RAM	1M	yes
rom	rom	2M	no
mbee	mbee On-board I2C EEPROM	32k	yes
ioee	ioee I/O Board II I2C EEPROM	NOT AVAILABLE	
i2c	i2c:bus:addr:bits[:hz] Generic access to I2C EEPROMS i2c buses (1): 0: "MB EEPROM" 1: "IO EEPROM" 2: "TS" 3: "Backplane"	-	yes

## See Also

[SRAM API \(p61\)](#)

[ROM API \(p81\)](#)

[Motherboard EEPROM API \(p187\)](#)

I2C API (p196)

IDPROM API (p151)

# Appendix 1

## CHANGES IN PREVIOUS VERSIONS

### A1.1 CHANGES IN V3.0.0

The key updates in v3.0 is the inclusion of the new event streaming API, which provides a single, simplified and unified mechanism to access changes to DirectPCI digital input devices, IDLP events, PFD events and other sources.

Support was added for these new motherboards:

- DPX-S430 (in v3.0.2.xxx)
- DPX-C710 (in v3.0.3.xxx) and systems with 8 intrusion inputs.
- DPX-E120 (in v3.0.1.xxx)

Support was added for Windows 7 OS with both 64- and 32-bit builds.

Support was added for 64-bit Linux builds.

- Support for new API functions
  - *dpci\_bios\_dump()*
  - *dpci\_get\_debug\_level()*
  - *dpci\_set\_debug\_level()*
  - *dpci\_ev\_wait\_all()*
  - *dpci\_ev\_wait()*
  - *dpci\_ev\_register\_callback()*
  - *dpci\_ev\_unregister\_callback()*
  - *dpci\_ev\_set\_debounce()*
  - *dpci\_io\_wait\_iport()*
  - *dpci\_io\_wait\_iports()*

- *dpci\_gpio\_read\_ip()*
- *dpci\_gpio\_write\_op()*
- *dpci\_gpio\_read\_op()*

## A1.2 CHANGES IN v2.3.0

The key updates in the v2.3.0 release are:

- Linux - Optional Password-protection for IDLP set-date and read event log, built into linux driver. Support for new API functions
  - *dpci\_id\_setdate\_psw()*
  - *dpci\_id\_readevent\_psw()*
- Windows - Moved netmos\_serial driver out of DirectPCI. Netmos serial driver is now available as a separate component.
- Performance improvements to IDLP firmware.

## A1.3 CHANGES IN v2.2.0

The key updates in the v2.2.0 release are:

- Support for new API functions for SRAM and ROM
  - *dpci\_sram\_map()*
  - *dpci\_sram\_unmap()*
  - *dpci\_rom\_map()*
  - *dpci\_rom\_unmap()*
- SRAM and ROM now available as block devices under windows
  - SRAM is available as B: and can be formatted to FAT
  - ROM is available as R:. ROM being a read-only device, cannot be formatted. But if the image is made up of a FAT format Files-system, it can be read in explorer.
- Performance improvements in SRAM and ROM memory access

### Changes in v2.0.0

The key updates in the v2.0.0 release are:

- Support for new API functions (and updated demonstration code) for batteries on boards with IDLP+
  - *dpci\_bat\_num\_batteries()*

- *dpci\_bat\_name()*
- *dpci\_bat\_number()*
- *dpci\_bat\_get\_status\_mask()*
- *dpci\_bat\_get\_level()*
- *dpci\_bat\_get\_status()*
- *dpci\_bat\_set\_check\_period()*
- *dpci\_bat\_set\_error\_level()*
- *dpci\_bat\_get\_errorlevel()*
- Support for new API functions (and updated demonstration code) for I/O port access
  - *dpci\_io\_change\_port()*
  - *dpci\_io\_read\_outport()*
- Support for new API function for I/O Board access
  - *dpci\_io\_board\_supported()*
- Deprecated functions for SRAM, ROM and battery support have been removed.
- ROM Extension detection function *dpci\_io\_isromext()* has been removed.
- SRAM and ROM APIs have been renamed with the suffix *dpci\_*
- Support for common APIs for error handling on Windows and Linux
  - *dpci\_last\_os\_error\_code()*
  - *dpci\_last\_os\_error\_string()*
  - *dpci\_os\_error\_string()*
- Support for new API functions for Quiet Mode settings on DPX S- and C-series boards
  - *dpci\_qm\_get()*
  - *dpci\_qm\_set()*

## Changes in v1.4.0

The key updates in the v1.4.0 release are:

- Support for the using One-wire and iButton devices via the GPIO facilities on the DPX-112 (r1.1), DPX-116 (r1.1) and DPX-117 boards.
- Two new command-line utilities *memutil* and *dpci* are provided and documented in this manual.
- Support for new API functions and (updated demonstration code):
  - *dpci\_io\_numiports()*
  - *dpci\_io\_numoports()*
  - *dpci\_e2\_size()*
  - *dpci\_i2c\_numbuses()*

- *dpci\_i2c\_bus\_name()*
- *dpci\_i2c\_bus\_number()*
- *dpci\_ts\_\**

## **Changes in v1.3.0**

The key updates in the v1.3.0 release are:

- Support for the DPX-112 and DPX-117 main boards
- Support for the new 80-0062 I/O Board II
- Support for the new 80-0063 Backplane II
- Enhanced support for I<sup>2</sup>C busses and peripherals.

No functions from previous releases have been deprecated in this release.

## **Changes in v1.2.0 and v1.2.1**

Some changes were made in previous releases of the API to ensure that the API is consistent between the Linux and the Microsoft Windows XP implementations and between different subsystems which share similar interfaces. Those functions replaced were marked clearly as deprecated.

Functions deprecated in v1.2.1 are no longer listed in this manual, although they will continue to function.

# **Appendix 2**

## **ADVANTECH WORLDWIDE**

Advantech Innocore has offices throughout the world. For a comprehensive list, please check our website at:

[www.advantech-innocore.com/contact/worldwide\\_office](http://www.advantech-innocore.com/contact/worldwide_office)

For international Sales representatives please check our website or email:

[sales@advantech-innocore.com](mailto:sales@advantech-innocore.com)