DSProfinet User's Guide

DSPROFINET
User's Guide
v1.1

4445 West 77th Street
Minneapolis, MN 55435
Phone: (952) 831-9556
FAX: (952) 831-4697

Contents

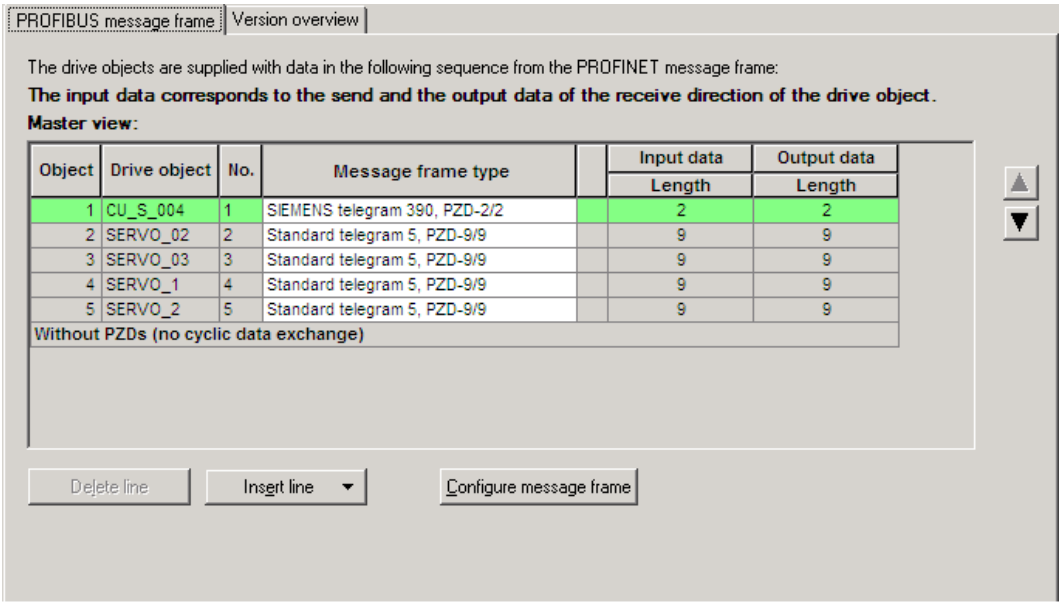
Read this first

A Quick Manual Overview

The DSProfinet is a powerful IRT Controller that transmits and receives the required real-time functions in a coordinated motion control application -- isochronously.

In its PCI form, this product is installed inside a PC that runs on a real-time operating system. This is because this product's dual port RAM should be fed with drive commands that are transmitted to the SINAMICS on a periodic basis.

The user of DSProfinet is assumed to have a prior knowledge of Siemens SINAMICS. The DSProfinet must be used only after he/she has configured the SINAMICS with STARTER Project. When you configure the cu320 with STARTER, part of what you will set is how many motors the control unit will be responsible for and which telegrams will be used for communicating with the cu320 and these motors. The DSProfinet IRT controller supports telegram 5 with SERVOs and, communicates with the cu320 using telegram 390 (below, the STARTER configuration window for four motors).



We have made every effort to make the use of this manual easy but in every step, we have assumed that the user has a full knowledge of drives and their respective controls.

Essentially, you must perceive the DSProfinet as a real-time conduit between your PC and the IRT network – a network comprising several drives to be controlled in an isochronous fashion.

The DSProfinet is accompanied by a CD that includes:

1) Three Starter project examples that we used to configure the cu320 unit for 2, 3 and 4 servo motors;

2) The C source codes – as an application example in DOS;

3) The files to be copied to SINAMICS flash. These files are the Profinet v2.1 firmware, along with a configuration for 2 motors.

The DSProfinet Configuration

DSProfinet IRT controller is configured in "C Program Example" listed in Section 9 of this manual. You must amend the configuration section of this code to the beginning of your application program. As you will see in this C program, the configuration information in dual-port RAM is comprised of the following:

Control Unit Selection

0x200	0x01 if only one cu310 online, 0x00 otherwise
0x201	0x00
0x202	0x01 if first of two cu310s is online, 0x00 otherwise
0x203	0x00
0x204	0x01 if second of two cu310s is online, 0x00 otherwise
0x205	0x00
0x206	0x01 if cu320 is online, 0x00 otherwise
0x207	0x00

Control Unit's IP address

0x208	0xc0 (192 for example)
0x209	0xa8 (168 for example)
0x20a	0x01 (1 for example)
0x20b	0xcb (203 for example)

Control Unit's MAC address

0x210	0x08 (for example)
0x211	0x00 (for example)
0x212	0x06 (for example)
0x213	0x93 (for example)
0x214	0xac (for example)
0x215	0xec (for example)

How many motors the control unit will be responsible for

0x21c	0x02 (for example)

After providing this information in dual-port RAM, the PC program needs to let the DSProfinet IRT controller know that it is present. First, clear the flag at 0x110 in dual-port RAM:

0x110	0x00
0x111	0x00

Now write the characters "config" to dual-port RAM, starting at 0x112:

0x112	0x63
0x113	0x6f
0x114	0x6e
0x115	0x66
0x116	0x69
0x117	0x67

1 Introduction to DSProfinet IRT Controller

Product Overview

DSPROFINET is a Profinet IRT Controller that functions over the time-tested Ethernet interface. By combining the power of Ethernet with Profinet IRT protocol that is both simple and reliable, a complete digital solution has been created for networking between motion control elements. The robustness of Ethernet's design is attested to by the fact that it continues to be adapted to new applications, and is constantly being upgraded to provide new capabilities.

Figure 1: DSPROFINET IRT Controller with Daisy-chained SINAMICS

When programming with DSPROFINET IRT Controller, a single Ethernet cable is sufficient to configure and program all devices (such as SINAMICS S120) on the Profinet IRT network.

Whether the Profinet network is inclusive of a single or multiple devices, DSPROFINET as the IRT Controller is capable of transmitting the isochronous real-time information through an Ethernet cable in a straight or daisy chained fashion. For example, the PROFIdrive commands on DSPROFINET links your PC program to multiple SINAMICS CU310s or a single CU320 unit in a coordinated system.

Determinism of the IRT Profinet

For high performance motion control applications, such as precise coordination of many motors with less than a microsecond delay between their coordinated commands, Profinet Controller is suited because it comes with an Isochronous Real-Time (IRT) channel. As indicated by the word "isochronous" in its acronym, Profinet IRT is used for closed-loop control of a servo system, where the control (both the set-point and feedback) for multiple devices occurs during the same sample period. This sample period can be as strict as 250 microseconds, meaning that the Controller in a Profinet IRT network issues its command to all devices every 250 microseconds. Similarly, each device in the Profinet IRT will respond with its data (for example, the actual position and/or speed in a motion system) during the same period.

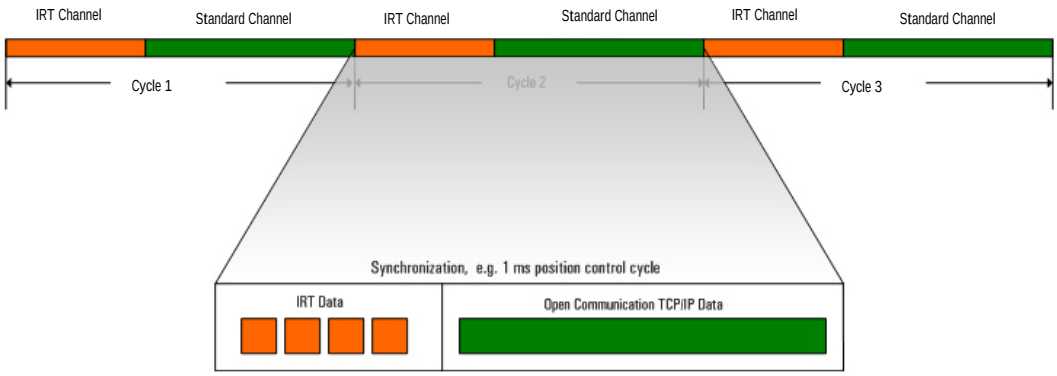


Figure 2: The IRT Channel in Network Cycle

Uniqueness of IRT Profinet Controller

Certainly other Ethernet protocols in motion control today operate on a regularly occurring interval basis. A relevant question may be: what is special about Profinet IRT Controller? The guiding factor that sets Profinet IRT apart from other real-time cyclic protocols is the concept of "jitter". The jitter is defined as a time fluctuation in the start of the interval. For example, in a one-millisecond sample period, if the controller started the next interval 100 nanoseconds after the termination of the previous one, the system could be described as having a jitter of 100 nanoseconds at this point in time.

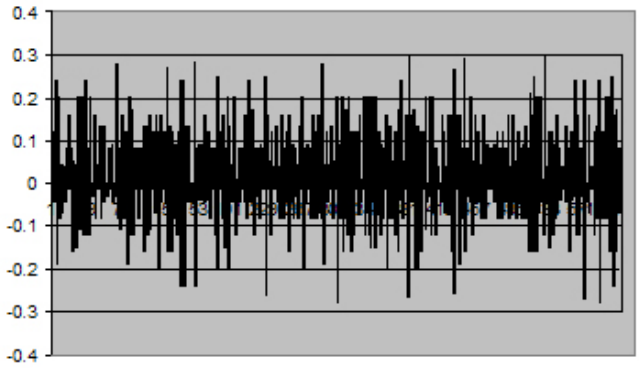


Figure 3: DSPROFINET Jitter in Microseconds vs. Sample Time

Other cyclic protocols may (EtherCat) or may not (Profinet RT, Ethernet PowerLink) be concerned with whether there is jitter at the start of each interval.

In the case of Profinet IRT, both devices and Controller are very concerned with jitter. The threshold for jitter allowed by the Profinet IRT protocol is defined to be one microsecond. Hence, an entity that wishes to serve as a controller in a Profinet IRT network must be able to start each cycle very precisely on the aforementioned millisecond boundary. The devices in a Profinet IRT network are designed to be made aware of when a controller is not adhering to the jitter requirement. Upon recognition of this situation, the devices will stop operating with the controller. It would then be up to the controller to essentially "start over" and show the devices that it is capable of operating within the jitter specification. (For example, maybe the controller wasn't able to start a series of cycles due to a bad cable. Once the cable is replaced, the controller will be able to attempt Profinet IRT with the devices from the beginning.)

The operation of cyclic control at these extremely precise intervals (such as one or two-millisecond interval times occurring within one microsecond of jitter) is what allows for extremely precise coordinated motion control applications to occur across multiple axes.

The Operational Principle behind IRT Controller

After an initial communication period between the controller and the device(s), Profinet IRT begins to start taking place. (Note that this initial communication period is just used to establish the parameters of the ensuing Profinet IRT communication, such as how much and what type of data will be exchanged during each interval, etc. It only needs to take place once and will last less than 30 seconds.)

There are two important classes of messages that get exchanged during each interval. (Occasionally there will be additional network management-type messages appearing in the network, but these are not related to control nor are they periodic. Also, they would certainly occur after the IRT messages for the current interval have been sent and received.) One of these classes of messages is synchronization messages, commonly referred to as "Sync" messages. These messages can be thought of as the "keep-alive" type of message. They are sent exclusively from the controller to the devices in the network and no response from the devices is necessary. Also, they do not contain any control data, but instead serve to ensure that the

controller is keeping up with the strict timing constraints of Profinet IRT. Namely, that it is starting the interval precisely one millisecond (or two milliseconds, as the case may be) from the last interval. This is the message that the devices will use to base the jitter calculations on. In other words, the Sync message is the message that must arrive within one microsecond of when it is supposed to, for every interval.

Also during the interval, messages carrying data from the device to the controller and the controller to the device will be exchanged. These messages are named Real-Time Class (RTC) messages.

For each interval, the controller will send out one RTC message for each device that it is controlling. This RTC message will contain the data, such as speed set points or position information that the device needs to have. In return, at the same time each device in the network is sending an RTC message to the controller. This RTC message will correspond with the RTC message it received. For example, if speed control is being performed, the controller's RTC to the device will have a desired speed (speed set point) and the device's RTC to the controller will have the actual speed.

2 DSProfinet IRT Controller & SINAMICS

Data Exchange in Between DSPROFINET and SINAMICS

When used with SINAMICS S120, DSPROFINET uses the PROFIdrive profile that contains "Dynamics Servo Control" (DSC) concept. This can be used to significantly increase the dynamic stability of the position control loop in what Siemens refers to as application class 4 with simple means. The telegrams used by DSPROFINET are 5 & 6 (for DSC 1 position encoder and DSC 2 position encoder respectively), 390, 391 and 392 (telegrams for control unit Drive Object 1, DO1, digital inputs / outputs). Cyclic communication is used to exchange time-critical process data.

Telegrams 5 & 6 (DSC motion parameters)

From the DSPROFINET IRT Controller to the SINAMICS S120 Telegram 5 (application class 4 DSC) delivers:

```
CTW1 .........      control word 1
NSOLL_B.......      32-bit speed set point
CTW2 .........      control word 2
G1_CTW........      encoder 1 control word
XERR .........      position deviation
```

returns the following data:

```
STW1............ control word 1
NIST_B ......... 32 bit actual speed
STW2 ........... status word 2
G1_STW ......... encoder 1 status word
G1_XIST1…..... encoder 1 actual position value 1
G1_XIST2…… …... encoder 1 actual position value 2
```

Telegram 390 (I/O parameters)

From the DSPROFINET IRT Profinet Controller to the SINAMICS S120 Telegram 390 delivers:

```
CU_CTW ..........control unit control word
O_DIGITAL .......16 bit digital output control word
```

Using the same telegrams S120 returns the following data to the DSPROFINET:

```
CU_STW ..........control unit status word
I_DIGITAL .......16 bit digital input control word
```

In addition, telegrams 391 and 392 also send and return (to DSPROFINET) probe status:

```
PR_CTW ..........from DSPROFINET to SINAMICS and
PR_STW ..........from SINAMICS to DSPROFINET
```

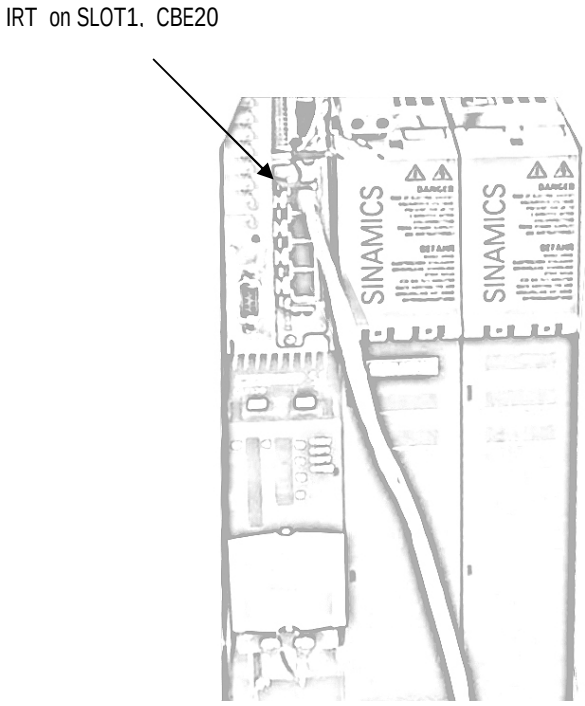


Figure 4: IRT cable connected to CU320

3 DSProfinet Hardware Platforms

DSPROFINET is offered in two platforms of stand-alone and PCI.

Stand-alone DSPROFINET

In a stand-alone form, DSPROFINET is powered by a single 5-volt power supply and it communicates with other subsystems via two RJ45 connectors J1 and J2.
J1 is linked to a standard Ethernet line on a laptop or other forms of PC, whereas J2 as the control link connects to other devices in an IRT Profinet system.

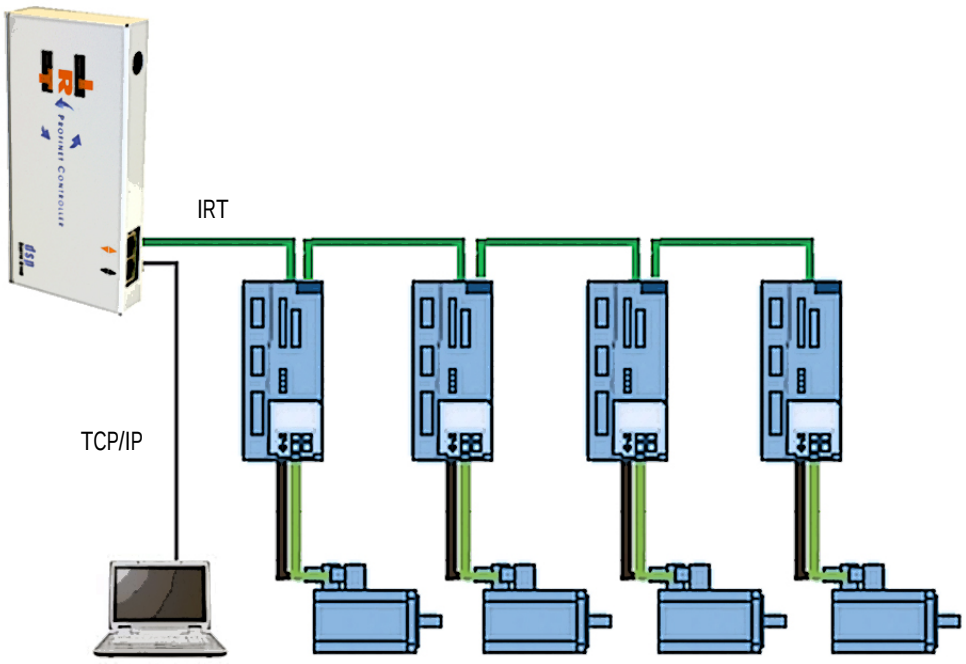


Figure 5: System Diagram For Stand-Alone DSPROFINET

PCI based DSPROFINET

A PCI based DSPROFINET resides in a PCI card slot of a PC.
The main difference between this one and the stand-alone unit is that in this unit PC information is transmitted to the DSPROFINET through the PCI bus.

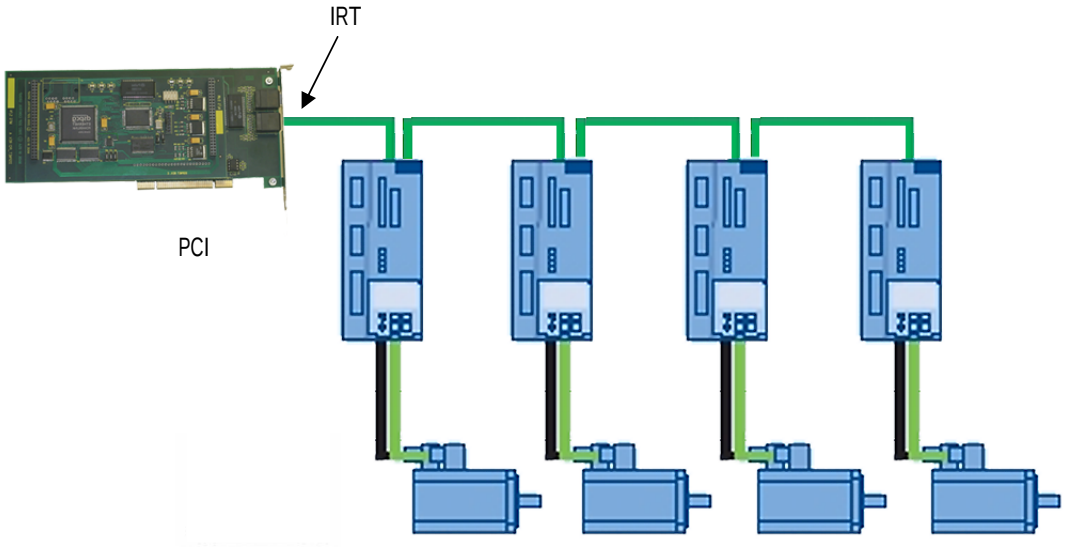


Figure 6: System Diagram For PC (PCI) Based DSPROFINET

The PCI based DSPROFINET is also unique in that its memory organization allows for buffering a group of commands and relieving the PC of attending to the drive every millisecond.

4 DSProfinet Memory Organization

DSProfinet Dual Port Ram Memory Organization

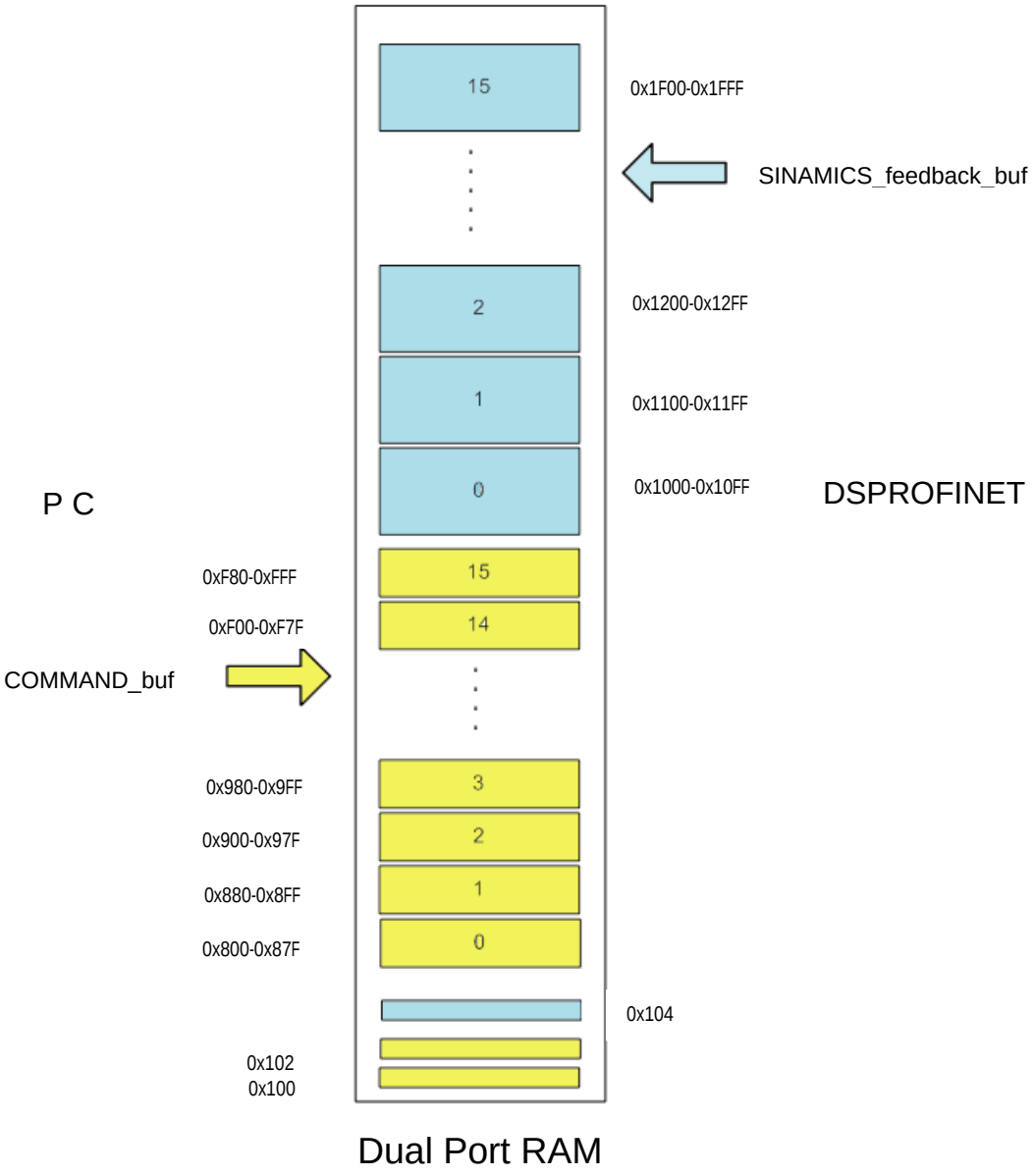


Figure 7: Command and SINAMICS Feedback Ring Buffers in Dual Port RAM

Note! The system of ring buffers with 16 cycle data blocks is necessary for CNC applications where advanced contouring is required. With a real-time OS however, most applications will not need more than one data point in each buffer. This one point consists of the data going to and coming back from the amplifiers in a single cycle.

Figure 7 illustrates the DSPROFINET's two ring buffers - each holding 16 Cycle Data Blocks or CDB points. The Command ring buffer holds the points to be transmitted to the SINAMICS and the SINAMICS buffer contains the feedback points that come back from the SINAMICS.

Command Buffer

Each Cycle Data Block (CDB) in the Command ring buffer is 128 bytes large and is placed in the following locations of the dual port ram:

CDB Point (Index)	Memory Location
15	0xF80 – 0xFFF
14	0xF00 – 0xF7F
13	0xE80 – 0xEFF
12	0xE00 – 0xE7F
11	0xD80 – 0xDFF
10	0xD00 – 0xD7F
9	0xC80 – 0xCFF
8	0xC00 – 0xC7F
7	0xB80 – 0xBFF
6	0xB00 – 0xB7F
5	0xA80 – 0xAFF
4	0xA00- 0xA7F
3	0x980 – 0x9FF
2	0x900 – 0x97F
1	0x880 – 0x8FF
0	0x800 – 0x87F

Command Buffer Index Location: 0x100 --- this location is set by DSPROFINET
Command Buffer Index Location: 0x102 --- optional location used by PC program

Table 1: memory organization for the command ring buffers

Location 0x100 in dual-port ram is read only and it holds the index to the Command buffer point that is available to DSPROFINET/SINAMICS for use in the most immediate cycle.

Location 0x102 in dual port ram is read/write and is modifiable by PC program only. The PC program can write to this location the Index that it will write to next. Writing to this location is optional – its only purpose is to prevent the use of data from 16 cycles ago (in case PC program can not stay ahead of the DSPROFINET and therefore the ring buffer rolls over to a point that is 16 cycles old). If the PC program can stay ahead of the DSPROFINET, this location doesn't need to be updated.

SINAMICS Feedback Buffer

Each CDB point in the SINAMICS feedback ring buffer is 256 bytes large and it holds the information that comes back from the SINAMICS.

CDB Point (Index)	Memory Location
15	0x1F00 – 0x1FFF
14	0x1E00 – 0x1EFF
13	0x1D00 – 0x1DFF
12	0x1C00 – 0x1CFF
11	0x1B00 – 0x1BFF
10	0x1A00 – 0x1AFF
9	0x1900 – 0x19FF
8	0x1800 – 0x18FF
7	0x1700 – 0x17FF
6	0x1600 – 0x16FF
5	0x1500 – 0x15FF
4	0x1400- 0x14FF
3	0x1300 – 0x13FF
2	0x1200 - 0x12FF
1	0x1100 – 0x11FF
0	0x1000 – 0x10FF

SINAMICS Feedback Buffer Index Location: 0x104

Table 2: memory organization for the SINAMICS feedback ring buffer

Location 0x104 is read only and it holds the index to the SINAMICS feedback buffer point that is going to be written to (by the DSPROFINET/SINAMICS) next. Therefore, if the value in 0x104 is 3, the PC program should read index location 2 to get the SINAMICS information.

5 ProfiDrive data Exchange

Data Exchange between DSPROFINET and SINAMICS

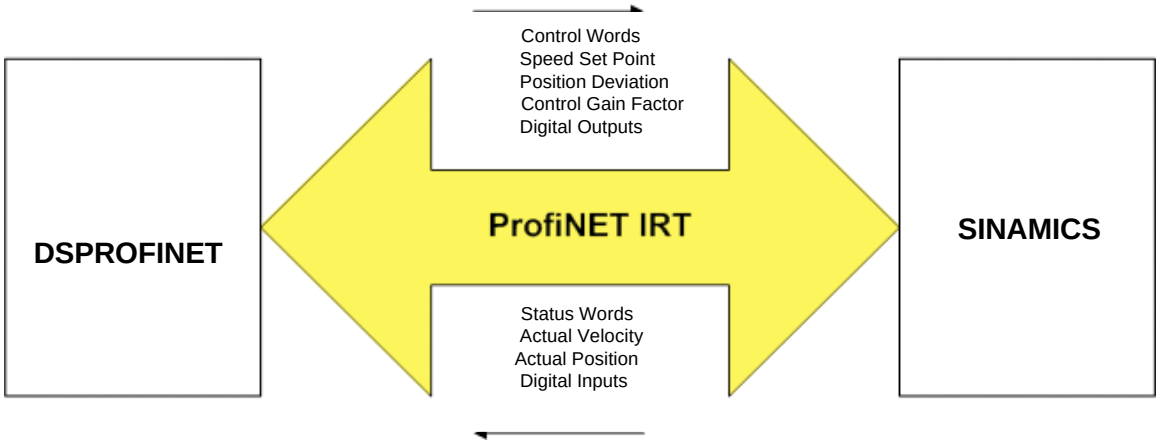


Figure 8: Data exchange between PC and SINAMICS

The data in the Command buffer consists of command codes and their respective arguments as tabulated below. Table 3 illustrates the contents of the Command buffer for four motors - using Telegrams 390 and 5.

BYTES	DESCRIPTION	
0 – 1	CU_CTW (control unit control word)	
2 – 3	O_DIGITAL (dig. output control word)	
4 – 5	CTW1 (control word 1)	MOTOR 1
6 – 9	NSOLL_B (32-bit speed set-point)	MOTOR 1
10-11	CTW2 (control word 2)	MOTOR 1
12-13	G1_CTW (encoder 1 control word)	MOTOR 1
14-17	XERR (position deviation)	MOTOR 1
18-21	KPC (position control gain factor)	MOTOR 1

22 – 23	CTW1 (control word 1)	MOTOR 2
24 – 27	NSOLL_B (32-bit speed set-point)	MOTOR 2
28 -29	CTW2 (control word 2)	MOTOR 2
30-31	G1_CTW (encoder 1 control word)	MOTOR 2
32-35	XERR (position deviation)	MOTOR 2
36-39	KPC (position control gain factor)	MOTOR 2

40 – 41	CTW1 (control word 1)	MOTOR 3
42 – 45	NSOLL_B (32-bit speed set-point)	MOTOR 3
46 – 47	CTW2 (control word 2)	MOTOR 3
48 -49	G1_CTW (encoder 1 control word)	MOTOR 3
50 – 53	XERR (position deviation)	MOTOR 3
54 – 57	KPC (position control gain factor)	MOTOR 3

58 – 59	CTW1 (control word 1)	MOTOR 4
60 – 63	NSOLL_B (32-bit speed set-point)	MOTOR 4
64 – 65	CTW2 (control word 2)	MOTOR 4
66 – 67	G1_CTW (encoder 1 control word)	MOTOR 4
68 – 71	XERR (position deviation)	MOTOR 4
72 – 75	KPC (position control gain factor)	MOTOR 4

Table 3: The contents of one data point in the Command buffer

The first 4 bytes of 128 (0-3) are for the control unit (CU310 or CU320/CBE20). The next 18 bytes (4-21) are for one motor and would be repeated once for each additional motor (up to 4).

Table 4 illustrates the contents of one CDB point in the SINAMICS feedback buffer.

BYTES	**DESCRIPTION**	
0 – 1	CU_STW (control unit status word)	
2 – 3	I_DIGITAL (dig. input control word)	
4 – 5	STW1 (status word 1)	MOTOR 1
6 – 9	NIST_B (32 bit actual speed)	MOTOR 1
10-11	STW2 (status word 2)	MOTOR 1
12-13	G1_STW (encoder 1 status word)	MOTOR 1
14-17	G1_XIST1 (encoder 1 act. Position val. 1)	MOTOR 1
18-21	G1_XIST2 (encoder 1 act. Position val. 2)	MOTOR 1

22 – 23	STW1 (status word 1)	MOTOR 2
24 – 27	NIST_B (32 bit actual speed)	MOTOR 2
28 -29	STW2 (status word 2)	MOTOR 2
30-31	G1_STW (encoder 1 status word)	MOTOR 2
32-35	G1_XIST1 (encoder 1 act. Position val. 1)	MOTOR 2
36-39	G1_XIST2 (encoder 1 act. Position val. 2)	MOTOR 2

40 – 41	STW1 (status word 1)	MOTOR 3
42 – 45	NIST_B (32 bit actual speed)	MOTOR 3
46 – 47	STW2 (status word 2)	MOTOR 3
48 -49	G1_STW (encoder 1 status word)	MOTOR 3
50 – 53	G1_XIST1 (encoder 1 act. Position val. 1)	MOTOR 3
54 – 57	G1_XIST2 (encoder 1 act. Position val. 2)	MOTOR 3

58 – 59	STW1 (status word 1)	MOTOR 4
60 – 63	NIST_B (32 bit actual speed)	MOTOR 4
64 – 65	STW2 (status word 2)	MOTOR 4
66 – 67	G1_STW (encoder 1 status word)	MOTOR 4
68 – 71	G1_XIST1 (encoder 1 act. Position val. 1)	MOTOR 4
72 – 75	G1_XIST2 (encoder 1 act. Position val. 2)	MOTOR 4

Table 4: The contents of one data point in the SINAMICS feedback buffer

The first 4 bytes of 256 (0-3) are for the control unit (CU310 or CU320/CBE20). The next 18 bytes (4-21) are for one motor and would be repeated once for each additional motor (up to 4).

6 Data Transfer To and From SINAMICS

Memory Access In Dual Port RAM

Data transfer from PC to DSPROFINET (and subsequently to SINAMICS)
DSPROFINET has a 16-bit word in its dual port RAM (memory location 0x100) that indicates which CDB (0-15) it will read next. The PC program can continue reading from this location to monitor which CDB index DSPROFINET is currently processing.

Once per cycle, DSPROFINET Performs the following functions:

1) Checks to see that the CDB index it wants to read is *not* the same one that the PC program wants to write to. The PC program can update dual-port RAM memory at memory location 0x102 with the index of the CDB it will write to next. If these indices are the same, DSPROFINET will use the same data that it used during the previous cycle. (For the case where there is no previous data, DSPROFINET will send a start instruction for control word 1, as well as 0x00 for the speed set-point, control word 2, encoder 1 control word, xerr, and kpc position control gain factor for each motor, as well as 0x0059a1b2 for the control unit's data.) It should be noted that the PC program should try and stay at least one point ahead of DSPROFINET, so for example when DSPROFINET wants to read CDB 0, the PC program will be ready to write to CDB 1.
2) Takes the data from the CDB and prepare to send it to the drive.
3) Updates the value in offset memory word 0x100 to reflect the next CDB index it will read from.

The PC program will fill in as many CDBs (indices 0-15) as it deems necessary, and then it will wait until it detects that an IRT connection has been made. It does this by reading 0x110 in dual-port RAM. DSPROFINET will write a 1 to 0x110 once IRT is established. Then the PC program can continue filling CDBs as DSPROFINET reads them, making sure not to "starve" DSPROFINET. It can do this by comparing where it is at to what is in memory offset 0x100, which is the CDB that DSPROFINET will be reading from next.

Data from SINAMICS to DSPROFINET (and subsequently to PC)
Each cycle will provide data for DSPROFINET to write to dual-port RAM. DSPROFINET will maintain a counter of which CDB in this buffer it intends to write to next. This will be at memory offset 0x104 in dual-port RAM. Upon receiving that data from the drive, DSPROFINET will write it to the appropriate slot and then update its counter at offset 0x104.

The PC program can read location 0x104 to see which CDB will be written to next by DSPROFINET. Based on this, it can read the previous CDB to get the most recent data.

7 ProfiDrive Instructions

ProfiDrive Instructions Used In Program Examples

The instruction set described below are those needed for Dynamic Servo Control (DSC) and I/O Operations, Telegrams 5 and 390 respectively. We will show a short description of each instruction and their respective telegrams. For more information, you may refer to the SINAMICS PROFIdrive manual.

Position control based on the velocity set-point interface with DSC.

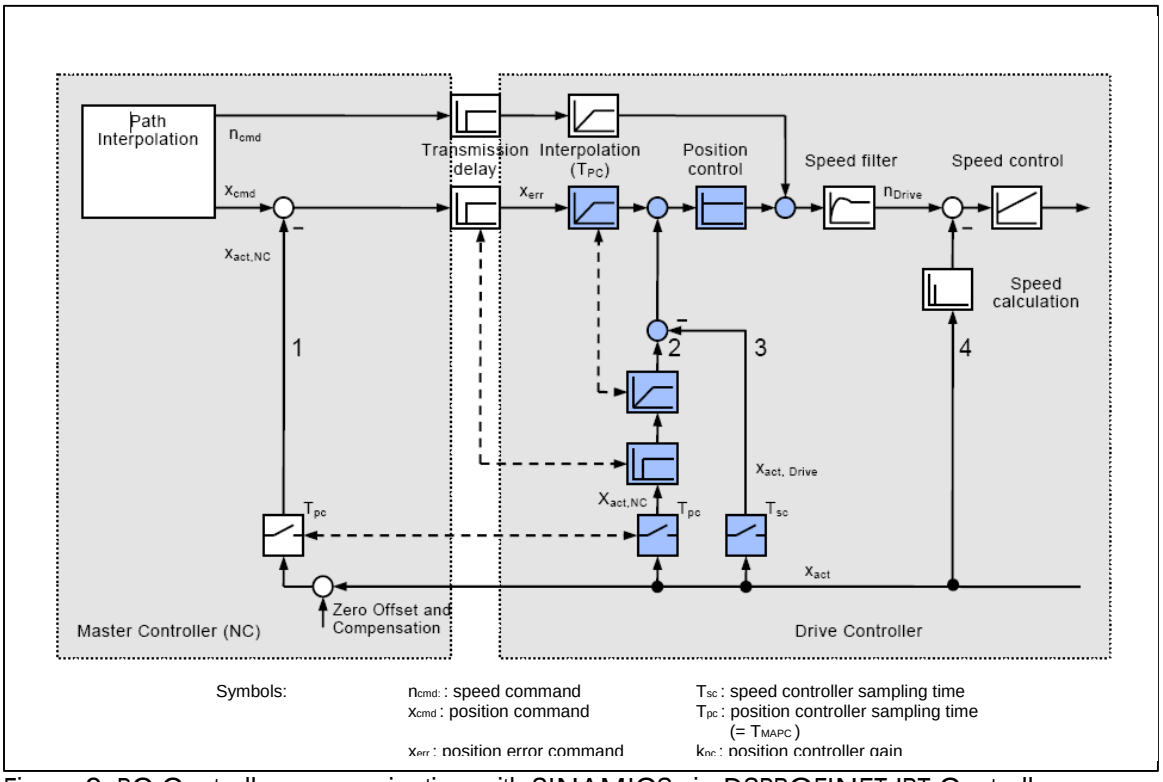


Figure 9: PC Controller communicating with SINAMICS via DSPROFINET IRT Controller

Telegram 5 - data going to the SINAMICS

CTW1 control word 1

This field is 2 bytes in size.

The value that is used can be either 0x047e (stop command) or 0x047f (start command).

For further information, please see the Siemens manual accompanying your drive.

NSOLL_B 32-bit speed set-point

This field is 4 bytes in size.

For further information, please see the Siemens manual accompanying your drive.

CTW2 control word 2

This field is 2 bytes in size.

For the first byte, we encode the PROFIdrive sign-of-life value. The PROFIdrive sign-of-life value is only 4 bits in size and must be the high nibble of the byte. The range of values for sign-of-life is 0x1 to 0xf. (For instance, we used the entire byte and sent 0x10, 0x20.... 0xf0, 0x10....)

The first byte is encoded as the PROFIdrive sign-of-life value and is therefore not modifiable.

For further information, please see the Siemens manual accompanying your drive.

G1_CTW encoder 1 control word

This field is 2 bytes in size.

We left this field as 0 the entire time.

For further information, please see the Siemens manual accompanying your drive.

XERR position deviation

This field is 4 bytes in size.

The value of this field should be updated each cycle. For our examples, we would perform a computation to determine what our ideal position was.

For further information, please see the Siemens manual accompanying your drive.

KPC **position control gain factor**

This field is 2 bytes in size.

For further information, please see the Siemens manual accompanying your drive.

Telegram 5 - data coming from the SINAMICS drive

STW1 **status word 1**

This field is 2 bytes in size.

For further information, please see the Siemens manual accompanying your drive.

NIST_B **32-bit actual speed**

This field is 4 bytes in size.

For further information, please see the Siemens manual accompanying your drive.

STW2 **status word 2**

This field is 2 bytes in size.

The drive uses the high nibble of the first byte to encode its own sign-of-life value. The values for the high nibble range from 0x1 to 0xf.

For further information, please see the Siemens manual accompanying your drive.

G1_STW **encoder 1 status word**

This field is 2 bytes in size.

For further information, please see the Siemens manual accompanying your drive.

G1_XIST1 **encoder 1 actual position value 1**

This field is 4 bytes in size.

For further information, please see the Siemens manual accompanying your drive.

G1_XIST2 **encoder 1 actual position value 2**

This field is 4 bytes in size.

For further information, please see the Siemens manual accompanying your drive.

Telegram 390 - data going to the SINAMICS drive

CU_CTW control unit control word

This field is 2 bytes in size.

The high nibble of the first byte is used for the PROFIdrive sign-of-life. In our application we use the entire first byte for it. We also set the Synchronization bit (bit 0) to 1.

For further information, please see the Siemens manual accompanying your drive.

O_DIGITAL 16-bit digital output control word

This field is 2 bytes in size.

The first byte is "Reserved." The second (low) byte is for digital outputs 8-15. As an example, if you had an LED connected to digital output 8 then setting the bit for digital output 8 to 1 would turn on the LED.

The example below is a screenshot from STARTER, the Siemens configuration tool. It shows digital output 8 (DO 8) as being configured as an output. Digital output 8 corresponds to the seventh "spot" in the X121 interface on the front of the drive.

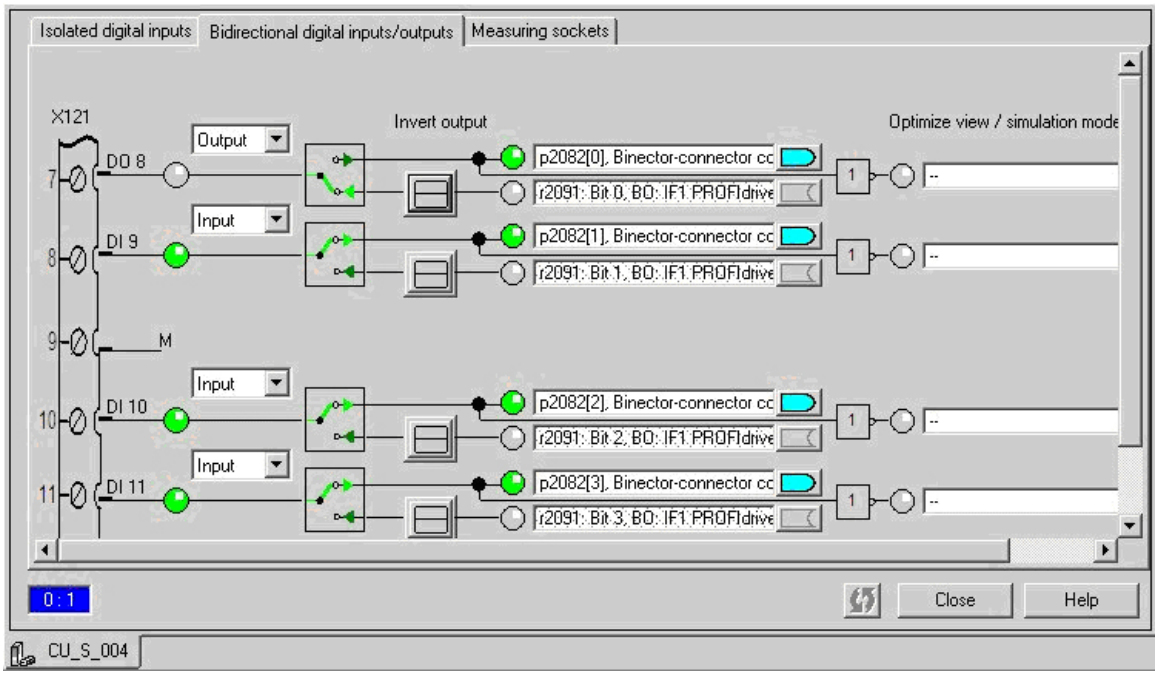


Figure 9: The STARTER picture of digital I/O on SINAMICS

For further information, please see the Siemens manual accompanying your drive.

Telegram 390 - data coming from the SINAMICS

`CU_STW  control unit status word`

This field is 2 bytes in size.

The high nibble of the first byte is used for the PROFIdrive sign-of-life.

For further information, please see the Siemens manual accompanying your drive.

`I_DIGITAL  16-bit digital input control word`

This field is 2 bytes in size.

The high byte is for digital inputs 0-7, and the low byte is for digital inputs 8-15. *For further information, please see the Siemens manual accompanying your drive.*

8 Application Programming

In this section we describe how your application program can interface with the DSPROFINET IRT Controller. The important concept to note is that your application will interact with the DSPROFINET IRT Controller via *dual-port ram*. This interaction will occur in a producer/consumer fashion. There are two ring buffers. For one ring buffer, your application *produces* data for controlling the motor(s) and the DSPROFINET IRT Controller *consumes* the data by sending it to the motor(s). The roles are reversed for the other ring buffer. The DSPROFINET IRT Controller will *produce* (or more accurately, take data produced by the motor(s)) and your application will *consume* the data.

First, let's discuss what data your application needs to produce. There are two distinct categories of data: data for the amplifier itself and data for the motor. Each will be described below.

Data for the SINAMICS drive

Each SINAMICS drive is to be given 4 bytes of data per cycle. For example, if the DSPROFINET IRT Controller is responsible for two cu310 amplifiers then your application will need to provide 8 bytes of SINAMICS data per cycle. If the DSPROFINET IRT Controller is responsible for one cu320 then your application will provide 4 bytes of SINAMICS data per cycle.

Data for the motors

Each motor is to be given 18 bytes of data per cycle.

Communicating with the DSPROFINET IRT Controller

As mentioned previously, communication is done via dual-port RAM (DPR). Writing to and reading from DPR can be achieved by using low level PCI functions found in the sample C programs provided.

The code examples included with this manual are written in C for the DOS operating system. Accesses to the PCI bus are made via DOS interrupt calls. DOS was chosen on purpose as the simplest real-time operating system available. Other platforms such as Windows CE and Linux would only obscure the principles we wish to convey.

The system calls required to establish communication with the DSPROFINET IRT Controller will vary depending on your operating system. Regardless of the operating system, however, you will begin by seeking the DSPROFINET IRT Controller's PCI unit number, based on its Manufacturer ID (10B5h) and its Function ID (5201h). Function `seek_device()`, included in file `PCI_LOW.C` in next section, shows how this is done in DOS, using the provided function `read_config()`.

Essentially, `seek_device()` checks the Manufacturer ID and Function ID for each PCI unit until it has either found the correct ID's or run out of units to check. `Read_config()` uses a DOS interrupt to access PCI configuration space. Again, in your application, you will use whatever means your operating system provides to seek PCI devices.

Once the DSPROFINET IRT Controller's PCI unit number has been located, `read_config()` is used again to find the address space at which the DSPROFINET IRT Controller's DPR is mapped on your PC. In `PCI_USR.C`, also in the appendix, in `main()`, variable dpr_off is assigned the 32-bit address at which DPR begins. The address is in the PCI unit's configuration space starting at offset 18h.

Once you have located the DSPROFINET IRT Controller's PCI unit number, and extracted the DPR starting address, you are ready to begin programming it.

When accessing the DSPROFINET IRT Controller's DPR via the PCI bus, it is important to keep in mind that DPR is 16 bits wide, whereas the PCI bus is 32 bits wide. The following illustration shows how DPR is mapped onto the PCI bus. In summary, the user will read or write 32 bits from or to the PCI bus. When reading, the high 16 bits need to be discarded by the user's PC application, and when writing the high 16 bits will be discarded by the DSPROFINET IRT Controller.

Keep in mind, too, that because of this, any access to 16 bits at DPR offset x will be made to 16 bits at PCI offset x * 2.

Reading data from dual port RAM via the PCI bus

Your PC application program needs to read data from the dual port RAM (an example of how it is done in DOS can be seen with `epeek()`, `epeek_wrapper()` and `byte_peek()` in the provided source code `pci_low.c`). Since the PCI bus is 32 bits wide and dual port ram is 16 bits wide, some mapping will have to occur, as seen in the following illustration.

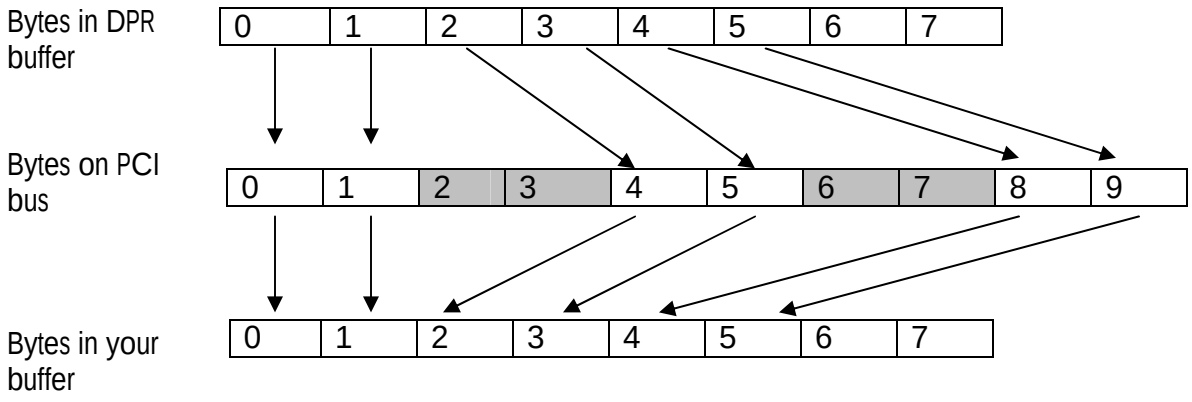


Figure 10: In order to get 8 bytes of meaningful data, the PCI bus will have to return 16 bytes total

The final mapping, from PCI bus to your buffer, is what is performed by `epeek_wrapper()` in our C code example.

Writing data to dual port RAM via the PCI bus

Your PC application program also will need to write data to dual port RAM (an example of how it is done in DOS can be seen with `epoke()` in the provided source code `pci_low.c.`) Since the PCI bus is 32 bits wide and dual port RAM is 16 bits wide, some mapping will have to occur as seen in the following illustration.

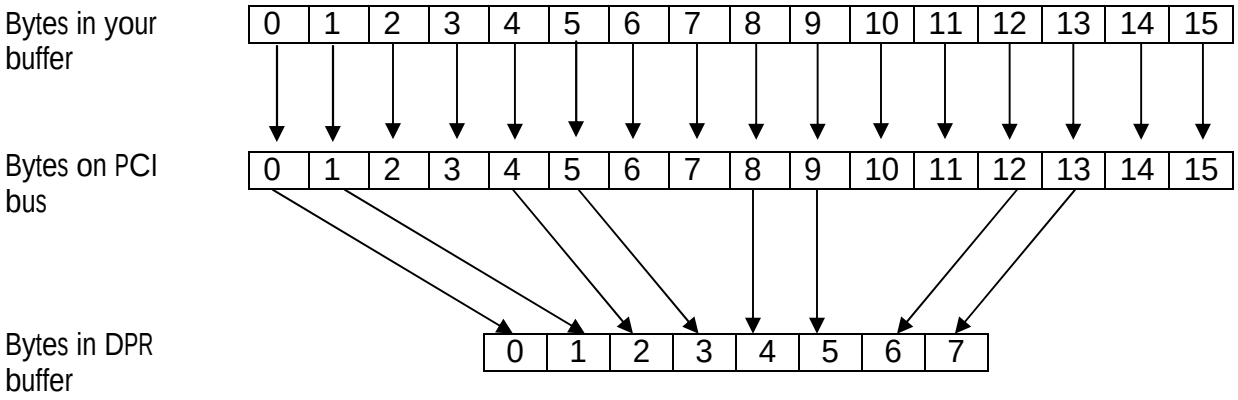


Figure 11: When writing to dual port RAM via PCI bus, half of the data you send will be discarded

Examples of creating data buffer for a 4-byte IP address and 6-byte MAC address, and writing them to dual port RAM can be seen in the example C code provided.

Dual-Port Ram Organization

As mentioned before, there are two buffers in the dual-port RAM for communication between your program application and the DSPROFINET IRT Controller – the Command and SINAMICS feedback buffers. The Command ring buffer will be for data that your application will use to control the motor. This data will be written by your application and sent to the amplifier. The SINAMICS feedback ring buffer will be for data that the drive produces and sends back to the DSPROFINET IRT Controller. Your application can read this data to "close the position loop". Each ring buffer will have 16 entries, thus supporting 16 cycles.

Command Ring Buffer – Data to the Amplifier:

> This ring buffer will begin at memory offset 0x800. Each slot (or Cycle Data Block) in this buffer will be 128 bytes, which is large enough to accommodate data for up to 4 motors.

SINAMICS Feedback Ring Buffer – Data from the Amplifier:

This ring buffer will begin at memory offset 0x1000. Each slot in this buffer will be 256 bytes, which is large enough to accommodate data from up to 4 motors.

Sample Application Program 1

This example will illustrate what would need to be done to control two motors with one cu320 unit with two motors.

First, since there is one cu320 and two motors, the DSPROFINET IRT Controller will require 4 + 18 + 18 = 40 bytes of data per cycle. We'll assume the data is stored in a variable called rtc and that your application has populated it with values:

```
unsigned char rtc[40];
rtc[0] = 0x00;
rtc[1] = 0x59;
…
…
…
rtc[39] = 0x10;
```

The next step is to determine which address in dual-port RAM to write to next. Your application will need to keep track of which slot (CDB index) to write to next:

```
int write_slot = 0;
```

Let's assume we should write at slot x, where 0<=x<=15. The calculation is as follows:

```
unsigned long slot_addr;//address to write to next
slot_addr = 0x800 + x * 128;
```

Where 0x800 is the base of the Command ring buffer and 128 is the size of one slot in this ring buffer.

Now your application must decide if it should write. (It might not be able to if it has gotten far enough ahead of the DSPROFINET IRT Controller such that it would start overwriting data that hasn't been read yet. Or, as in the DOS program example provided, your application may decide to only write a new cycle after the previous cycle has been read.) The DSPROFINET IRT Controller will update its location in the buffer each time it does a read. It will read first and then update - for example, it will read slot 0 and then update it's pointer to slot 1, indicating that the data has been taken from slot 0. Your application program can now write to slot0 again (assuming that it has already written slots 1,2,3, ..., 15 and therefore was waiting on slot 0 to come open) or slot 1 (if it is just trying to stay one step ahead of the DSPROFINET IRT controller).

```
data = byte_peek(dpr_off,0x100);
while ((int)data != write_slot) {
      data = byte_peek(dpr_off,0x100);
```

```
}
```

The condition in the while loop means that the DSPROFINET IRT Controller has not read the data in the slot before `write_slot`. Where `write_slot` is the CDB that your application is ready to write to next. Once it reads the data, it will update 0x100 with the number of the next slot.

Sample Application Program 2

In addition to controlling the motor only, let's assume you wish to read back the actual position or velocity (or both) from the motor. For this we will use the second ring buffer.

For this buffer, the DSPROFINET IRT Controller performs a write first, and then your application can read.

Let's assume you have a cu320 amplifier controlling 2 motors. There will be 4 bytes of data coming back for the cu320 and 18 bytes of data coming back for *each* motor.

```
unsigned char rtc[40];        //4 + 2*18 = 40
```

Address 0x104 in dual-port RAM will be the index of the slot that the DSPROFINET IRT controller will write to next. Therefore, your application can read the latest data by going to the previous slot:

```
int read_slot = 0;
data = byte_peek(dpr_off,0x104);
if ((int)data == 0)
      read_slot = 15;
else
      read_slot = (int)data – 1;

//Each slot is size 0x100.  The base is at 0x1000.
```

Now you can use epeek() or one of it's variants (see `pci_low.c` in section 9) to extract the data yourself, or you can use the functions provided, such as `get_actual_pos1()` or `get_actual_speed()` to retrieve individual fields.

9 C Program Examples

The following C files are included in the DSPROFINET CD:

1. `pci_low.c` that contains functions for PCI access to the DSPROFINET IRT controller and;
2. `pci_usr.c` that contains a main application for controlling motors with the DSPROFINET IRT controller - it makes use of the functions in `pci_low.c` to do things such as read from and write to the dual-port RAM. We now discuss the main points of each of the two files.

1. pci_low.c – PCI access to DSPROFINET IRT controller's dp RAM

The two most important operations to perform over the PCI bus are reading the dual-port RAM and writing to it. The functions `epeek()`and `epoke()`perform these tasks. Other "wrapper" functions, such as `epeek_wrapper()` and `byte_peek()` have been provided for illustration purposes, but these functions rely on `epeek()` itself.

1.1 Functions for reading data from the dual-port RAM

The following three functions can be used to read data from dual-port RAM. Note that the PCI bus is 32 bits, whereas the dual-port RAM of the DSPROFINET IRT controller is 16 bits. This means that every time 16 bits are read, 32 bits will be returned to the host PC application. The extra 16 bits will all be zeros. The function `epeek()` will leave the zeros in place

`epeek()`

The function `epeek()` will take in an address, a storage buffer, and the number of 16-bit words desired. Starting from the address provided, it will read in the number of words asked for and store them in the provided storage buffer, along with the extra padding of 16 bits of zeros for every 16 bit word read. Therefore the buffer provided must be twice as large as the number of words asked for.

`epeek_wrapper()`

The function `epeek_wrapper()`, like `epeek()`, will take in an address, a storage buffer, and the number of 16-bit words desired. Starting from the address provided, it will read in the number of words asked for and store them in the provided storage buffer. It makes use of the `epeek()`function to get the data, hence storage must also be twice as large as the number of words asked for. The extra feature of `epeek_wrapper` is that it illustrates how to compact the data to the front of the buffer.

byte_peek()
The function byte_peek() will take in the address for the beginning of dual-port RAM and the byte number from that address that is desired. This byte will then be returned. This function can be regarded as a wrapper function of epeek() for just getting an individual byte from dual-port RAM. It will mask off everything else and simply return 1 byte.

1.2 Functions for writing data to dual-port RAM

epoke()
The function epoke() will take in an address to write to, a buffer of data to write and, the number of words to write. It will then write the specified number of words from the buffer to the provided address.

1.3 Other PCI functions are:

read_config()
This function will take in a register number. It will read a word at that register from the PCI configuration space. This is how to get information about the PCI device, such as manufacturer information or the physical address of the dual-port RAM.

write_config()
This function will take in a word of data and a register number in the PCI configuration space. It will write the given word to the register.

seek_device()
This function will locate the PCI device with manufacturer ID 10b5h and function ID 5201h (the DSPROFINET IRT controller). It will return the device number.

2. pci_usr.c – Controlling multiple motors via PCI based DSProfinet IRT controller

main()
The first task of main() is to get the address of the dual-port RAM from the perspective of the PCI. Next, configuration information such as which type of control unit (cu310 or cu320) is online, the control unit's IP address, the control unit's MAC address and, the number of motors to be controlled need to be written to the dual-port RAM. Also, main() clears the flags and then lets the DSProfinet IRT controller to know the configuration information is present.

The configuration information will be in dual-port RAM as follows:

Control Unit Selection

0x200	0x01 if only one cu310 online, 0x00 otherwise
0x201	0x00
0x202	0x01 if first of two cu310s is online, 0x00 otherwise
0x203	0x00
0x204	0x01 if second of two cu310s is online, 0x00 otherwise
0x205	0x00
0x206	0x01 if cu320 is online, 0x00 otherwise
0x207	0x00

Control Unit's IP address

0x208	0xc0 (192 for example)
0x209	0xa8 (168 for example)
0x20a	0x01 (1 for example)
0x20b	0xcb (203 for example)

Control Unit's MAC address

0x210	0x08 (for example)
0x211	0x00 (for example)
0x212	0x06 (for example)
0x213	0x93 (for example)
0x214	0xac (for example)
0x215	0xec (for example)

How many motors the control unit will be responsible for

0x21c	0x02 (for example)

After providing this information in dual-port RAM, the PC program needs to let the DSProfinet IRT controller know that it is present. First, clear the flag at 0x110 in dual-port RAM:

0x110	0x00
0x111	0x00

Now write the characters "config" to dual-port RAM, starting at 0x112:

0x112	0x63
0x113	0x6f
0x114	0x6e
0x115	0x66
0x116	0x69
0x117	0x67

After that, `main()` will simply need to write and read data to and from the dual-port RAM. User needs to at least insert one Cycle Data Block (CDB) in advance of an IRT connection.

`two_motor_points()`

This function will generate the points to control the motion of two motors.

It is assume that the STARTER project you use with this example has appropriately configured a cu320 for two motors on a v2.1 Profinet firmware.

three_motor_points()
This function will generate the points to control the motion of three motors.

It is assume that the STARTER project you use with this example has appropriately configured a cu320 for three motors on a v2.1 Profinet firmware.

four_motor_points()
This function will generate the points to control the motion of four motors.

It is assume that the STARTER project you use with this example has appropriately configured a cu320 for four motors on a v2.1 Profinet firmware.

get_cu_stw()
This function will retrieve the 2-byte control unit status word field.

get_i_digital()
This function will retrieve the 2-byte digital input control word field.

get_stw1()
This will retrieve the 2-byte status word 1 field for a given motor (1-4).

get_actual_speed()
This will retrieve the 32-bit actual speed field for a given motor (1-4).

get_stw2()
This will retrieve the 2-byte status word 2 field for a given motor (1-4).

get_g1_stw()
This will retrieve the 2-byte encoder 1 status word field for a given motor (1-4).

get_actual_pos1()
This function will retrieve the 4-byte encoder 1 actual position value 1 field for a given motor (1-4).

get_actual_pos2()
This function will retrieve the 4-byte encoder 1 actual position value 2 field for a given motor (1-4).

PCI_LOW.C Listing

```
/*********************************************************************
 * This file contains the implementation of lower-level routines for
 * accessing dual-port RAM on the DSProfinet IRT controller.
 *********************************************************************/
#include<stdio.h>
#include<conio.h>
#include<dos.h>
#include<math.h>

/*********************************************************************
 *
 * The following interrupt descriptions are copied from Ralf Brown's
 * PC interrupt list, available at http://www.cs.cmu.edu/~ralf/files.html.
 *
 * The Borland C function int86() is used to execute these interrupts.
 * Most DOS C compilers have an equivalent function, although syntax
 * will probably differ somewhat.
 *
 * Functions epeek() and epoke() utilize the following standard PC
 * system interrupt.  Essentially, peripherals on the PCI bus are
 * mapped into extended memory, so this is all we need to access them,
 * once we know the addresses at which they are located.
 *
 * --------B-1587-------------------------------
 * INT 15 - SYSTEM - COPY EXTENDED MEMORY
 *      AH = 87h
 *      CX = number of words to copy (max 8000h)
 *      ES:SI -> global descriptor table (see #00499)
 *       Return: CF set on error
 *          CF clear if successful
 *          AH = status (see #00498)
 * Notes: copy is done in protected mode with interrupts disabled by
 *        the default BIOS handler; many 386 memory managers perform
 *        the copy with interrupts enabled on the PS/2 30-286 &
 *        "Tortuga" this function does not use the port 92h for A20
 *        control, but instead uses the keyboard controller (8042).
 *        Reportedly this may cause the system to crash when access
 *        to the 8042 is disabled in password server mode (see also
 *        PORT 0064h,#P0398) this function is incompatible with the
 *        OS/2 compatibility box
 *
 * SeeAlso: AH=88h,AH=89h,INT 1F/AH=90h
 *
 * (Table 00498)
 * Values for extended-memory copy status:
 *      00h     source copied into destination
 *      01h     parity error
 *      02h     interrupt error
 *      03h     address line 20 gating failed
 *      80h     invalid command (PC,PCjr)
 *      86h     unsupported function (XT,PS30)
 *
 * Format of global descriptor table:
 * Offset       Size            Description    (Table 00499)
 * 00h          16 BYTEs        zeros (used by BIOS)
 * 10h          WORD            source segment length in bytes (2*CX-1 or greater)
 * 12h          3 BYTEs         24-bit linear source address, low byte first
 * 15h          BYTE            source segment access rights (93h)
 * 16h          WORD            (286) zero
 *                              (386+) extended access rights and high byte of source
 *                               address
 * 18h          WORD            destination segment length in bytes (2*CX-1 or greater)
 * 1Ah          3 BYTEs         24-bit linear destination address, low byte first
 * 1Dh          BYTE            destination segment access rights (93h)
 * 1Eh          WORD            (286) zero
 *                              (386+) extended access rights and high byte of destin.
```

```
 *                            address
 * 20h         16 BYTEs       zeros (used by BIOS to build CS and SS descriptors)
 *
 *
 *
 * Function read_config() uses the following interrupt to read the config
 * space for the PCI device specified by registers BH and BL.
 *
 * --------X-1AB109-----------------------------
 * INT 1A - PCI BIOS v2.0c+ - READ CONFIGURATION WORD
 *      AX = B109h
 *      BH = bus number
 *      BL = device/function number (bits 7-3 device, bits 2-0 function)
 *      DI = register number (0000h-00FFh, must be multiple of 2) (see #00878)
 * Return: CF clear if successful
 *          CX = word read
 *      CF set on error
 *      AH = status (00h,87h) (see #00729)
 *      EAX, EBX, ECX, and EDX may be modified
 *      all other flags (except IF) may be modified
 * Notes: this function may require up to 1024 byte of stack; it will not
 *        enable interrupts if they were disabled before making the call
 *         the meanings of BL and BH on entry were exchanged between the
 *        initial drafts of the specification and final implementation
 *
 * BUG:   the Award BIOS 4.51PG (dated 05/24/96) incorrectly returns FFFFh
 *        for register 00h if the PCI function number is nonzero
 *
 *
 *
 * Function write_config() uses the following interrupt to write the config
 * space for the PCI device specified by registers BH and BL.
 *
 * --------X-1AB10C-----------------------------
 * INT 1A - PCI BIOS v2.0c+ - WRITE CONFIGURATION WORD
 *      AX = B10Ch
 *      BH = bus number
 *      BL = device/function number (bits 7-3 device, bits 2-0 function)
 *      DI = register number (multiple of 2 less than 0100h)
 *      CX = word to write
 * Return: CF clear if successful
 *      CF set on error
 *      AH = status (00h,87h) (see #00729)
 *      EAX, EBX, ECX, and EDX may be modified
 *      all other flags (except IF) may be modified
 * Notes: this function may require up to 1024 byte of stack; it will not
 *        enable interrupts if they were disabled before making the call
 *        the meanings of BL and BH on entry were exchanged between the
 *        initial drafts of the specification and final implementation
 ***********************************************************************/

//The base address for dual-port RAM, from the PCI's perspective.
extern unsigned long dpr_off;

/***********************************************************************
 * epeek()
 *
 * Read num_words of data at address peek_add into the buffer pointed
 * to by peek_data.
 ***********************************************************************/
void epeek(unsigned long peek_add, unsigned int *peek_data,
           unsigned long num_words)
{
        //union REGS is defined in dos.h.  For every register on an
        //8086 compatible processor, this union has a corresponding
        //entry.  The function int86 expects two structures like this,
        //one containing the values to load into all of the processor's
        //registers before launching an interrupt (inregs), and one in
        //which to store the values of the registers after the interrupt
```

```
        //has completed (outregs), for the user to review.
        union REGS inregs, outregs;

        //The global descriptor table from Table 00499 in interrupt
        //description above.
        char gdt[ 48 ];

        //The address (in 32-bit format) at which we wish to store the
        //data which we will read from the PCI device.
        unsigned long add_data;


        //Figure address of user-supplied data buffer.  We need to
        //convert from the 8086's 20-bit segment:offset format to
        //a regular 32-bit offset.
        add_data = FP_SEG( peek_data );      //Load the segment into add_data

        //Shift the segment up four bits to convert it to an offset,
        //and add the address offset.
        add_data = ( add_data << 4 ) + FP_OFF( peek_data );

        //Figure number of BYTES to read back from memory.
        num_words *= 2;

        //Set up global descriptor table (see Table 0499 in interrupt
        //description above).
        gdt[ 0x10 ] = num_words;
        gdt[ 0x11 ] = num_words >> 8;
        gdt[ 0x12 ] = peek_add;
        gdt[ 0x13 ] = peek_add >> 8;
        gdt[ 0x14 ] = peek_add >> 16;
        gdt[ 0x15 ] = 0x93;
        gdt[ 0x16 ] = ( num_words >> 16 ) & 0x0f;
        gdt[ 0x17 ] = peek_add >> 24;
        gdt[ 0x18 ] = num_words;
        gdt[ 0x19 ] = num_words >> 8;
        gdt[ 0x1a ] = add_data;
        gdt[ 0x1b ] = add_data >> 8;
        gdt[ 0x1c ] = add_data >> 16;
        gdt[ 0x1d ] = 0x93;
        gdt[ 0x1e ] = ( num_words >> 16 ) & 0x0f;
        gdt[ 0x1f ] = add_data >> 24;

        //Set up interrupt.
        inregs.h.ah = 0x87;           //Interrupt function: move block.
        inregs.x.cx = num_words / 2;  //No. of words to copy.

        //Store offset of global descriptor table into segment ES
        //in register SI.
        inregs.x.si = FP_OFF( &gdt[ 0 ] );

        //Note that segment register ES is set by default to the segment
        //in which our gdt is located, so we do not need to set it.

        //Execute interrupt.
        int86( 0x15, &inregs, &outregs );
}

/*********************************************************************
 * epeek_wrapper()
 *
 * @param addr        The offset into dpr.
 * @param data        Storage for what is found at offset 'addr'.
 * @param num_chars   The number of bytes to get.
 *
 * This function is used to illustrate the handling of data coming
 * back when you request two or more bytes from dual-port RAM.  For
 * every 2 legitimate bytes that PCI returns to us, there will be 2
 * bytes of 0x00 immediately following.  Hence if you requested 4
```

```
 * bytes, it would come back like this:
 *
 * Byte 0: byte0
 * Byte 1: byte1
 * Byte 2: 0x00
 * Byte 3: 0x00
 * Byte 4: byte2
 * Byte 5: byte3
 * Byte 6: 0x00
 * Byte 7: 0x00
 *
 * This function will rectify the data so it is presented to the
 * caller as the caller would expect:
 *
 * Byte 0: byte0
 * Byte 1: byte1
 * Byte 2: byte2
 * Byte 3: byte3
 ********************************************************************/
void epeek_wrapper(unsigned long addr, unsigned char * data, int num_chars)
{
        int i;

        //Even though epeek expects the number of words, we can safely
        //pass it the number of chars.  This is because for every word
        //that it finds, it brings along a word of 0x0000 padding, which
        //we will discard.
        epeek(dpr_off + addr*2, (unsigned int *)data, num_chars);

        //Now half of what came back is the padding with zeros, so
        //lets remove those.  The pattern is this:
        //Indexes 0,1, 4,5, 8,9, etc. have the legitimate data and
        //the other indexes have 0x00.
        for (i=0; i < num_chars; i+=2) {
                data[i] = data[i*2];
                data[i + 1] = data[(i*2) + 1];
        }
}

/********************************************************************
 * epoke()
 *
 * Write num_words of data to address poke_add from the buffer pointed
 * to by poke_data.
 ********************************************************************/
void epoke(unsigned long poke_add, unsigned int *poke_data,
           unsigned long num_words)
{
        //union REGS is defined in dos.h.  For every register on an
        //8086 compatible processor, this union has a corresponding
        //entry.  The function int86 expects two structures like this,
        //one containing the values to load into all of the processor's
        //registers before launching an interrupt (inregs), and
        //one in which to store the values of the registers after
        //the interrupt has completed (outregs), for the user to review.
        union REGS inregs, outregs;

        //The global descriptor table from Table 00499 in interrupt
        //description above.
        char gdt[ 48 ];

        //The address (in 32-bit format) from which we will read the
        //data which we will write to the PCI device.
        unsigned long add_data;


        //Figure address of user-supplied data buffer.  We need to
        //convert from the 8086's 20-bit segment:offset format to
        //a regular 32-bit offset.
```

```
        add_data = FP_SEG( poke_data );       //Load the segment into add_data.

        //Shift the segment up four bits to convert it to an offset,
        //and add the address offset.
        add_data = ( add_data << 4 ) + FP_OFF( poke_data );

        //Figure number of BYTES to copy into memory.
        num_words *= 2;

        //Set up global descriptor table (see Table 0499 in interrupt
        //description above).
        gdt[ 0x10 ] = num_words;
        gdt[ 0x11 ] = num_words >> 8;
        gdt[ 0x12 ] = add_data;
        gdt[ 0x13 ] = add_data >> 8;
        gdt[ 0x14 ] = add_data >> 16;
        gdt[ 0x15 ] = 0x93;
        gdt[ 0x16 ] = ( num_words >> 16 ) & 0x0f;
        gdt[ 0x17 ] = add_data >> 24;
        gdt[ 0x18 ] = num_words;
        gdt[ 0x19 ] = num_words >> 8;
        gdt[ 0x1a ] = poke_add;
        gdt[ 0x1b ] = poke_add >> 8;
        gdt[ 0x1c ] = poke_add >> 16;
        gdt[ 0x1d ] = 0x93;
        gdt[ 0x1e ] = ( num_words >> 16 ) & 0x0f;
        gdt[ 0x1f ] = poke_add >> 24;

        //Set up interrupt.
        inregs.h.ah = 0x87;           //Interrupt function: move block.
        inregs.x.cx = num_words / 2;  // No. of words to copy.

        //Store offset of global descriptor table into segment ES
        //in register SI.
        inregs.x.si = FP_OFF( &gdt[ 0 ] );

        //Note that segment register ES is set by default to the segment
        //in which our gdt is located, so we do not need to set it.

        //Execute interrupt.
        int86( 0x15, &inregs, &outregs );
}

/*********************************************************************
 * read_config()
 *
 * Read a word at register reg_number from the PCI configuration space
 * for the unit specified by inregs.h.bl.
 *********************************************************************/
unsigned int read_config(unsigned int reg_number, unsigned char device)
{
        union REGS inregs, outregs;

        inregs.h.ah = 0xb1;    //Interrupt function: Read configuration word.
        inregs.h.al = 0x09;
        inregs.h.bh = 0x00;    //Bus number; may be 0 or 1, depending on your motherboard.
        inregs.h.bl = device;  //Device number; generally, each PCI slot gets its own
device number.
        inregs.x.di = reg_number; //Register number; PCI devices have a variety of
registers we can read.
        int86( 0x1a, &inregs, &outregs );

        return( outregs.x.cx );
}

/*********************************************************************
 * write_config()
 *
 * Write the word (data) to register reg_number in the PCI
```

```
 * configuration space for the unit specified by inregs.h.bl.
 ****************************************************************/
unsigned int write_config(unsigned int reg_number, unsigned int data,
                          unsigned char device)
{
        union REGS inregs, outregs;

        inregs.h.ah = 0xb1;    //Interrupt function: Write configuration word.
        inregs.h.al = 0x0c;
        inregs.h.bh = 0x00;    //Bus number; may be 0 or 1, depending on your motherboard.
        inregs.h.bl = device;  //Device number; generally, each PCI slot gets its own
device number.
        inregs.x.di = reg_number; //Register number; PCI devices have a variety of
registers we can write.
        inregs.x.cx = data;    //Data to write.

        //Execute interrupt.
        int86( 0x1a, &inregs, &outregs );

        return( outregs.x.cflag );
}

/****************************************************************
 * seek_device()
 *
 * Finds the PCI device with manufacturer ID 10b5h and function ID
 * 5201h.  Returns the device number.
 ****************************************************************/
unsigned char seek_device( void )
{
        unsigned char device;
        union REGS inregs, outregs;


        for( device = 0x00; device < 0xff && ( read_config( 0, device ) != 0x10B5
        || read_config( 2, device ) != 0x5201 ); device += 0x01 );

        return( device );
}

/****************************************************************
 * byte_peek()
 *
 * @param DPR_off      The beginning of DPR
 * @param byte_num     The number of the byte in DPR that
 *                     should be returned.  For example, 5
 *                     means the fifth byte from the
 *                     beginning.
 *
 * A wrapper function to handle getting an individual byte from DPR.
 ****************************************************************/
unsigned char byte_peek(unsigned long DPR_off, unsigned long byte_num) {
        unsigned long data[ 1 ];

        if (byte_num % 2) {
                epeek(DPR_off + ( ((byte_num - 1)/2) * 4), (unsigned int *) data, 1);
                return (data[0] >> 8) & 0xff;
        }
        else {
                epeek(DPR_off + ( (byte_num/2) * 4), (unsigned int *) data, 1);
                return (data[0]     ) & 0xff;
        }
}
```

PCI_USR.C Program Listing

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>
#include<math.h>

//pci_low.c contains the implementation of lower-level PCI functions.
#include "pci_low.c"

//The base address for dual-port RAM, from the PCI's perspective.
unsigned long dpr_off;

//The buffer for one cycle's points.  4 + 18 + 18 = 40 is enough if
//there are just two motors, add 18 bytes for each additional motor.
unsigned char rtc[76]; //4 motors

//We need to keep track of which slot we write to next.
int write_slot = 0;

//We need to keep track of which slot to read from.  The DSProfinet IRT
//controller will place the data coming from the cu320 in one of the 16
//slots (256 bytes each, starting at 0x1000) in dual-port RAM.
int read_slot = 0;

//Just so we can see XERR changing
int xerr_inc = 1;

//As part of the development process, it helps to only run for a
//certain number of cycles.
int num_run_cycles = 10000;

//The buffer for the PCI to use.  It will be the rtc[] buffer
//with every 16-bit word padded with zeros to make them 32 bits.
//This is because the PCI bus will read out 32 bits at a time, but
//only 16 of those go to the 16-bit dual-port RAM.  The other 16,
//which will now be zeros, will be lost.
unsigned char pci_buf[152] = {        //twice as large as rtc[]
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
};

//The buffer holding the control unit's MAC address.  As with pci_buf[],
//every two bytes will be padded with 0x0000 so that when PCI takes 32 bits,
//the 16 that are lost are the 0x0000 bits.  The mac address that I am using
//here is 0x08 0x00 0x06 0x93 0xac 0xec
unsigned char mac_addr[12] = {        //twice as large as the needed 6 bytes
        0x08,0x00,0x00,0x00,0x06,0x93,0x00,0x00,0xac,0xec,0x00,0x00
};

//The buffer holding the control unit's IP address.
//192.168.1.203
unsigned char ip_addr[8] = {
        0xc0,0xa8,0x00,0x00,0x01,0xcb,0x00,0x00
};

//The buffer holding information on which type of control unit is online.
```

```
//Each of 0x200, 0x202, 0x204, 0x206 can be 0 or 1, depending on your
//configuration:
//      0x200 = 0x00;  //standalone cu310
//      0x201 = 0x00;
//      0x202 = 0x00;  //daisy-chain head cu310
//      0x203 = 0x00;
//      0x204 = 0x00;  //daisy-chain tail cu310
//      0x205 = 0x00;
//      0x206 = 0x01;  //cu320 - 0x01 means this is online
//      0x207 = 0x00;
unsigned char cu_selection[16] = {
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x00
};

//The buffer holding information on how many motors the control unit will
//be responsible for.  This can be 0x02, 0x03, or 0x04.
unsigned char num_motors[4] = {
        0x02,0x00,0x00,0x00
};

//A buffer holding all zeros, used for clearing specific addresses.
unsigned char clear_buf[4] = {
        0x00,0x00,0x00,0x00
};

//The buffer holding the string "config".  Writing this to 0x112 will let
//the DSProfinet IRT controller know that configuration info is present.
//This will allow it to start making an IRT connection.
unsigned char cfg_ready_buf[12] = {
        0x63,0x6f,0x00,0x00,0x6e,0x66,0x00,0x00,0x69,0x67,0x00,0x00
};

//Generates the points to move two motors.
void two_motor_points(void);

//Generates the points to move three motors.
void three_motor_points(void);

//Generates the points to move four motors.
void four_motor_points(void);

//Functions for retrieving individual fields from the data sent by the
//control unit (cu320) to the DSProfinet IRT controller.  get_cu_stw() and
//get_i_digital() return data about the control unit itself and hence take
//no parameters.  The remaining functions return data about a motor and
//therefore take in the motor number as a parameter.  The data starts at
//offset 0x1000 in dual-port RAM.
unsigned int get_cu_stw(void);              //2 bytes, control unit status word
unsigned int get_i_digital(void);    //2 bytes, digital input control word
unsigned int get_stw1(int);          //2 bytes, status word 1
unsigned long get_actual_speed(int); //4 bytes, actual speed
unsigned int get_stw2(int);          //2 bytes, status word 2
unsigned int get_g1_stw(int);        //2 bytes, encoder 1 status word
unsigned long get_actual_pos1(int);  //4 bytes, actual pos. value 1
unsigned long get_actual_pos2(int);  //4 bytes, actual pos. value 2

void main()
{
        unsigned int data;
        unsigned char b_data;
        unsigned char device;
        unsigned char input;  //What the user typed in
        int i=0;
        unsigned long m1_actual_pos1 = 0, m2_actual_pos1 = 0;
        unsigned long m1_actual_speed = 0, m2_actual_speed = 0;
        unsigned long m1_datas[5000], m2_datas[5000];
        int index = 0;
        FILE * fp;
```

```
clrscr();

//Seek the PCI ethernet device on the PCI bus and display its
//device number.
printf( "DEVICE: %04xh\n", seek_device() );
device = seek_device();

//Get back manufacturer/device info from the PCI ethernet device.
printf( "device = %04xh, Board ID: %04x %04x %04x %04xh\n", device,
                read_config( 0, device ), read_config( 2, device ),
                read_config( 4, device ), read_config( 6, device ));

//Get the address of dual-port RAM in PCI space.  The low nibble
//is config stuff that should be stripped out if it's nonzero.
dpr_off = read_config(0x1a, device) * 65536 +
         read_config(0x18, device);

////////////////////////////////////////////////////////////////
//The first thing that needs to be done is to set the
//configuration in dual-port RAM.
////////////////////////////////////////////////////////////////

//Write the control unit selection information to dpr, starting
//at address 0x200.
epoke(dpr_off + (0x200*2), (unsigned int *)cu_selection, 8);

//Write the IP address, starting at 0x208 in dpr.
epoke(dpr_off + (0x208*2), (unsigned int *)ip_addr, 4);

//Write the MAC address, starting at 0x210 in dpr.
epoke(dpr_off + (0x210*2), (unsigned int *)mac_addr, 6);

//Write the number of motors to control to dpr.
epoke(dpr_off + (0x21c*2), (unsigned int *)num_motors, 2);

//In addition, we need to clear any flags that the DSProfinet IRT
//controller sets for us.  If the controller was restarted (but not
//repowered) the flag indicating a valid IRT connection may still
//be set.  We clear this, and as the board comes up it will set it
//again.
epoke(dpr_off + (0x110*2), (unsigned int *)clear_buf, 2);

//Let the DSProfinet IRT controller know that configuration info
//is present.  This will allow it to start making an IRT connection.
epoke(dpr_off + (0x112*2), (unsigned int *)cfg_ready_buf, 6);

printf("\nPress 's' to start: ");
scanf("%c", &input);

//We have to write the first cycle's data in advance of the
//DSProfinet IRT controller getting a connection, so data is
//there by the time DSProfinet gets to IRT.  After that we will
//wait on the DSProfinet IRT controller before doing subsequent
//writes.  The following operations are commented in great
//detail in the while loop below.
two_motor_points();   //Generate the points for two motors.
for (i=0; i< 80; i+=4) {
       pci_buf[i]     = rtc[i/2];
       pci_buf[i+1]   = rtc[i/2 + 1];
}
epoke(dpr_off + (0x800*2), (unsigned int *)pci_buf, 40);
write_slot = 1;
printf("The first cycle's points are ready.\n");

//Before we can do cyclic data exchange with the motor, the
//DSProfinet IRT controller must have a connection with the
//control unit.
b_data = byte_peek(dpr_off, 0x110);
while (b_data != 0x01)
```

```
        b_data = byte_peek(dpr_off, 0x110);
    printf("IRT connection with control unit is ready!\n");

    //IRT Application.  Instead of having a 1ms interrupt on this
    //end and trying to synchronize it with the DSProfinet IRT
    //controller's interrupt, we can key off of when the DSProfinet
    //IRT controller does something.  For instance, we need to write
    //points to the buffer at 0x800.  I can write one cycle's worth
    //of points there, wait until the DSProfinet IRT controller
    //reads them and sets 0x100.  (0x100 changing is my proof that
    //the DSProfinet IRT controller read data.)  Then I can write
    //again, etc.

//  while (1) {
    while (num_run_cycles) {
        ////////////////////////////////////////////////////
        //Fill in my internal buffer, i.e generate one cycle's
        //points (and store them in rtc[]) that I will send
        //later.
        ////////////////////////////////////////////////////
        two_motor_points();

        //The PCI bus will take 32 bits at a time, but only
        //16 of those will make it to dual-port RAM (since
        //it's a 16-bit RAM).  Therefore we need to pad
        //every 16-bit word with 16 bits of zeros so that
        //the zeros will be what gets lost.  In this case,
        //pci_buf[] was initialized with 0x00s and so we
        //just need to fill in the data we want, leaving
        //gaps of zeros every 16 bits.  So it will look
        //like this:
        //
        //pci_buf[0]   = rtc[0]
        //pci_buf[1]   = rtc[1]
        //pci_buf[2]   = 0x00
        //pci_buf[3]   = 0x00
        //
        //pci_buf[4]   = rtc[2]
        //pci_buf[5]   = rtc[3]
        //pci_buf[6]   = 0x00
        //pci_buf[7]   = 0x00
        //...
        //...
        //...
        //pci_buf[76]  = rtc[38]
        //pci_buf[77]  = rtc[39]
        //pci_buf[78]  = 0x00
        //pci_buf[79]  = 0x00
        for (i=0; i< 80; i+=4) {
            pci_buf[i]     = rtc[i/2];
            pci_buf[i+1]   = rtc[i/2 + 1];
        }

        ////////////////////////////////////////////////////
        //Wait for an indication that the DSProfinet IRT
        //controller has read the previous data we sent to it.
        //The value at 0x100 will be the number of the slot that
        //the DSProfinet IRT controller will read from next,
        //which is the one we want to write to now.
        ////////////////////////////////////////////////////
        b_data = byte_peek(dpr_off, 0x100);
        while ((int)b_data != write_slot) {
            //The DSProfinet IRT controller hasn't read the
            //last data yet.  So we keep checking.  Once the
            //DSProfinet IRT controller HAS read it, we will
            //write new data (knowing that it'll be a
            //millisecond or so before the new data gets
            //read).  Basically, when the DSProfinet IRT
            //controller sets 0x100 it is telling us what
```

```
                        //slot it will read during the next cycle.
                        b_data = byte_peek(dpr_off, 0x100);
                }

                ////////////////////////////////////////////////////
                //Now the DSProfinet IRT controller has read the last
                //set of points we gave it.  Write the next points.  The
                //base of the slots that the Profinet IRT board reads
                //from to get data for the drive is at offset 0x800.
                //Each slot is 128 bytes.
                ////////////////////////////////////////////////////
                epoke(dpr_off + (0x800 + write_slot*128)*2, (unsigned int *)pci_buf, 40);
                write_slot++;
                if (write_slot == 16)
                        write_slot = 0;

                ////////////////////////////////////////////////////
                //Get what the control unit sent.  By reading offset
                //0x104 we can find where the DSProfinet IRT controller
                //will write to next; therefore, we read from the
                //previous slot to get the most current data.
                ////////////////////////////////////////////////////
                b_data = byte_peek(dpr_off, 0x104);
                if ( (int)b_data == 0)
                        read_slot = 15;
                else
                        read_slot = (int)b_data - 1;

                //Get the data...
                m1_actual_pos1 = get_actual_pos1(1);  //for motor 1
                m2_actual_pos1 = get_actual_pos1(2);  //for motor 2
                m1_actual_speed = get_actual_speed(1);
                m2_actual_speed = get_actual_speed(2);

                //Use the data as necessary...I will store it and
                //later print to a file for verification purposes.
                if (index < 5000) {
                        m1_datas[index] = m1_actual_speed;
                        m2_datas[index] = m2_actual_speed;
                        index++;
                }


                num_run_cycles--;
        }

        //Being as I am able to stay one step ahead of the DSProfinet
        //IRT controller, I haven't been updating 0x102.  But now,
        //if I set it to slot 0, eventually the DSProfinet IRT
        //controller will notice, assume that it caught up to me,
        //assume that the data at slot 0 hasn't been updated (in other
        //words, that the data at slot 0 is 16 cycles old) and will
        //send the previous cycle's data.
        epoke(dpr_off + (0x102*2), (unsigned int *)clear_buf, 2);

        fp = fopen("motor1.txt", "w");
        for (index=0; index < 5000; index++)
                //fprintf(fp, "0x%08lx\n", m1_datas[index]);
                fprintf(fp, "%lu\n", m1_datas[index]);
        fclose(fp);

        fp = fopen("motor2.txt", "w");
        for (index=0; index < 5000; index++)
                fprintf(fp, "%lu\n", m2_datas[index]);
        fclose(fp);
}

/*********************************************************************
 * two_motor_points()
```

```
 *
 * This will generate the points to move two motors.  The
 * data can be placed in one slot of the buffer in dual-port RAM.
 ************************************************************************/
void two_motor_points(void)
{
        unsigned long xerr = 0;

        //TELEGRAM 390.  Note that the DSProfinet IRT controller will
        //overwrite the first byte, so just send 0x00.
        rtc[0] = 0x00;
        rtc[1] = 0x59;
        rtc[2] = 0xa1;
        rtc[3] = 0xb2;

        ////////////////////////////////////////////////////
        // First Telegram 5 - this is for motor1
        ////////////////////////////////////////////////////
        //Control Word 1
        rtc[4] = 0x04;
        rtc[5] = 0x7f;

        //Speed Setpoint
        rtc[6] = 0x00;
        rtc[7] = 0xff;//0x00;
        rtc[8] = 0xff;//0x00;
        rtc[9] = 0xff;//0x00;

        //Control Word 2
        //Note that the DSProfinet IRT controller will overwrite
        //the first byte of this word, so just send 0x00.
        rtc[10] = 0x00;
        rtc[11] = 0x00;

        //G1_STW
        rtc[12] = 0x00;
        rtc[13] = 0x00;

        //XERR
        xerr = 10000 + xerr_inc;
        xerr_inc++;
        if (xerr_inc == 10000)
                xerr_inc= 0;
        rtc[14] = (xerr >> 24) & 0x000000ff;
        rtc[15] = (xerr >> 16) & 0x000000ff;
        rtc[16] = (xerr >>  8) & 0x000000ff;
        rtc[17] = (xerr) & 0x000000ff;

        //KPC Gains
        rtc[18] = 0x00;
        rtc[19] = 0x00;
        rtc[20] = 0x00;//0x27;
        rtc[21] = 0x00;//0x10;

        ////////////////////////////////////////////////////
        // Second Telegram 5 - this is for motor2
        ////////////////////////////////////////////////////
        //Control Word 1
        rtc[22] = 0x04;
        rtc[23] = 0x7f;

        //Speed Setpoint
        rtc[24] = 0x01;//0x00;
        rtc[25] = 0xff;//0x00;
        rtc[26] = 0xff;//0x00;
        rtc[27] = 0xff;//0x00;

        //Control Word 2
        //Note that the DSProfinet IRT controller will overwrite
```

```
        //the first byte of this word, so just send 0x00.
        rtc[28] = 0x00;
        rtc[29] = 0x00;

        //G1_STW
        rtc[30] = 0x00;
        rtc[31] = 0x00;

        //XERR
        xerr = 30000;
        rtc[32] = (xerr >> 24) & 0x000000ff;
        rtc[33] = (xerr >> 16) & 0x000000ff;
        rtc[34] = (xerr >>  8) & 0x000000ff;
        rtc[35] = (xerr) & 0x000000ff;

        //KPC Gains
        rtc[36] = 0x00;
        rtc[37] = 0x00;
        rtc[38] = 0x00;//0x27;
        rtc[39] = 0x00;//0x10;
}

/*********************************************************************
 * three_motor_points()
 *
 * This will generate the points to move three motors.  The data can
 * be placed in one slot of the buffer in dual-port RAM.
 *********************************************************************/
void three_motor_points(void)
{
        unsigned long xerr = 0;

        //TELEGRAM 390.  Note that the DSProfinet IRT controller will
        //overwrite the first byte, so just send 0x00.
        rtc[0] = 0x00;
        rtc[1] = 0x59;
        rtc[2] = 0xa1;
        rtc[3] = 0xb2;

        ////////////////////////////////////////////////////
        // First Telegram 5 - this is for motor1
        ////////////////////////////////////////////////////
        //Control Word 1
        rtc[4] = 0x04;
        rtc[5] = 0x7f;

        //Speed Setpoint
        rtc[6] = 0x00;
        rtc[7] = 0xff;//0x00;
        rtc[8] = 0xff;//0x00;
        rtc[9] = 0xff;//0x00;

        //Control Word 2
        //Note that the DSProfinet IRT controller will overwrite
        //the first byte of this word, so just send 0x00.
        rtc[10] = 0x00;
        rtc[11] = 0x00;

        //G1_STW
        rtc[12] = 0x00;
        rtc[13] = 0x00;

        //XERR
        xerr = 10000 + xerr_inc;
        xerr_inc++;
        if (xerr_inc == 10000)
                xerr_inc= 0;
        rtc[14] = (xerr >> 24) & 0x000000ff;
        rtc[15] = (xerr >> 16) & 0x000000ff;
```

```
rtc[16] = (xerr >>  8) & 0x000000ff;
rtc[17] = (xerr) & 0x000000ff;

//KPC Gains
rtc[18] = 0x00;
rtc[19] = 0x00;
rtc[20] = 0x00;//0x27;
rtc[21] = 0x00;//0x10;

////////////////////////////////////////////////////
// Second Telegram 5 - this is for motor2
////////////////////////////////////////////////////
//Control Word 1
rtc[22] = 0x04;
rtc[23] = 0x7f;

//Speed Setpoint
rtc[24] = 0x01;//0x00;
rtc[25] = 0xff;//0x00;
rtc[26] = 0xff;//0x00;
rtc[27] = 0xff;//0x00;

//Control Word 2
//Note that the DSProfinet IRT controller will overwrite
//the first byte of this word, so just send 0x00.
rtc[28] = 0x00;
rtc[29] = 0x00;

//G1_STW
rtc[30] = 0x00;
rtc[31] = 0x00;

//XERR
xerr = 30000;
rtc[32] = (xerr >> 24) & 0x000000ff;
rtc[33] = (xerr >> 16) & 0x000000ff;
rtc[34] = (xerr >>  8) & 0x000000ff;
rtc[35] = (xerr) & 0x000000ff;

//KPC Gains
rtc[36] = 0x00;
rtc[37] = 0x00;
rtc[38] = 0x00;//0x27;
rtc[39] = 0x00;//0x10;

////////////////////////////////////////////////////
// Third Telegram 5 - this is for motor3
////////////////////////////////////////////////////
//Control Word 1
rtc[40] = 0x04;
rtc[41] = 0x7f;

//Speed Setpoint
rtc[42] = 0x00;
rtc[43] = 0xff;
rtc[44] = 0xff;
rtc[45] = 0xff;

//Control Word 2
//Note that the DSProfinet IRT controller will overwrite
//the first byte of this word, so just send 0x00.
rtc[46] = 0x00;
rtc[47] = 0x00;

//G1_STW
rtc[48] = 0x00;
rtc[49] = 0x00;

//XERR
```

```
        xerr = 40000;
        rtc[50] = (xerr >> 24) & 0x000000ff;
        rtc[51] = (xerr >> 16) & 0x000000ff;
        rtc[52] = (xerr >>  8) & 0x000000ff;
        rtc[53] = (xerr) & 0x000000ff;

        //KPC Gains
        rtc[54] = 0x00;
        rtc[55] = 0x00;
        rtc[56] = 0x27;
        rtc[57] = 0x10;
}

/*********************************************************************
 * four_motor_points()
 *
 * This will generate the points to move four motors.  The data can be
 * placed in one slot of the buffer in dual-port RAM.
 *********************************************************************/
void four_motor_points(void)
{
        unsigned long xerr = 0;

        //TELEGRAM 390.  Note that the DSProfinet IRT controller will
        //overwrite the first byte, so just send 0x00.
        rtc[0] = 0x00;
        rtc[1] = 0x59;
        rtc[2] = 0xa1;
        rtc[3] = 0xb2;

        ////////////////////////////////////////////////////
        // First Telegram 5 - this is for motor1
        ////////////////////////////////////////////////////
        //Control Word 1
        rtc[4] = 0x04;
        rtc[5] = 0x7f;

        //Speed Setpoint
        rtc[6] = 0x00;
        rtc[7] = 0xff;
        rtc[8] = 0xff;
        rtc[9] = 0xff;

        //Control Word 2
        //Note that the DSProfinet IRT controller will overwrite
        //the first byte of this word, so just send 0x00.
        rtc[10] = 0x00;
        rtc[11] = 0x00;

        //G1_STW
        rtc[12] = 0x00;
        rtc[13] = 0x00;

        //XERR
        xerr = 10000 + xerr_inc;
        xerr_inc++;
        if (xerr_inc == 10000)
                xerr_inc= 0;
        rtc[14] = (xerr >> 24) & 0x000000ff;
        rtc[15] = (xerr >> 16) & 0x000000ff;
        rtc[16] = (xerr >>  8) & 0x000000ff;
        rtc[17] = (xerr) & 0x000000ff;

        //KPC Gains
        rtc[18] = 0x00;
        rtc[19] = 0x00;
        rtc[20] = 0x27;
        rtc[21] = 0x10;
```

```
//////////////////////////////////////////////////////
// Second Telegram 5 - this is for motor2
//////////////////////////////////////////////////////
//Control Word 1
rtc[22] = 0x04;
rtc[23] = 0x7f;

//Speed Setpoint
rtc[24] = 0x00;
rtc[25] = 0x00;
rtc[26] = 0x00;
rtc[27] = 0x00;

//Control Word 2
//Note that the DSProfinet IRT controller will overwrite
//the first byte of this word, so just send 0x00.
rtc[28] = 0x00;
rtc[29] = 0x00;

//G1_STW
rtc[30] = 0x00;
rtc[31] = 0x00;

//XERR
xerr = 30000;
rtc[32] = (xerr >> 24) & 0x000000ff;
rtc[33] = (xerr >> 16) & 0x000000ff;
rtc[34] = (xerr >>  8) & 0x000000ff;
rtc[35] = (xerr) & 0x000000ff;

//KPC Gains
rtc[36] = 0x00;
rtc[37] = 0x00;
rtc[38] = 0x27;
rtc[39] = 0x10;

//////////////////////////////////////////////////////
// Third Telegram 5 - this is for motor3
//////////////////////////////////////////////////////
//Control Word 1
rtc[40] = 0x04;
rtc[41] = 0x7f;

//Speed Setpoint
rtc[42] = 0x00;
rtc[43] = 0x00;
rtc[44] = 0x00;
rtc[45] = 0x00;

//Control Word 2
//Note that the DSProfinet IRT controller will overwrite
//the first byte of this word, so just send 0x00.
rtc[46] = 0x00;
rtc[47] = 0x00;

//G1_STW
rtc[48] = 0x00;
rtc[49] = 0x00;

//XERR
xerr = 15000;
rtc[50] = (xerr >> 24) & 0x000000ff;
rtc[51] = (xerr >> 16) & 0x000000ff;
rtc[52] = (xerr >>  8) & 0x000000ff;
rtc[53] = (xerr) & 0x000000ff;

//KPC Gains
rtc[54] = 0x00;
rtc[55] = 0x00;
```

```
        rtc[56] = 0x27;
        rtc[57] = 0x10;

        ////////////////////////////////////////////////////
        // Fourth Telegram 5 - this is for motor4
        ////////////////////////////////////////////////////
        //Control Word 1
        rtc[58] = 0x04;
        rtc[59] = 0x7f;

        //Speed Setpoint
        rtc[60] = 0x00;
        rtc[61] = 0xff;
        rtc[62] = 0xff;
        rtc[63] = 0xff;

        //Control Word 2
        //Note that the DSProfinet IRT controller will overwrite
        //the first byte of this word, so just send 0x00.
        rtc[64] = 0x00;
        rtc[65] = 0x00;

        //G1_STW
        rtc[66] = 0x00;
        rtc[67] = 0x00;

        //XERR
        xerr = 25000;
        rtc[68] = (xerr >> 24) & 0x000000ff;
        rtc[69] = (xerr >> 16) & 0x000000ff;
        rtc[70] = (xerr >>  8) & 0x000000ff;
        rtc[71] = (xerr) & 0x000000ff;

        //KPC Gains
        rtc[72] = 0x00;
        rtc[73] = 0x00;
        rtc[74] = 0x27;
        rtc[75] = 0x10;
}

/*********************************************************************
 * get_cu_stw()
 *
 * @return cu_stw     2-byte control unit status word field
 *
 * Retrieve the control unit status word field from the data sent by
 * the control unit to the DSProfinet IRT controller.
 *
 * The data starts at the beginning of the CDB (slot).
 *********************************************************************/
unsigned int get_cu_stw()
{
        unsigned int cu_stw = 0;

        //Since we have a 16-bit RAM but a 32-bit PCI, for every 16
        //bits that get returned there will be 16 bits of 0x000 padding,
        //hence you must create storage for twice as many chars.
        unsigned char data[4]; //we only wany 2 chars.

        epeek_wrapper(0x1000 + read_slot*256, data, 2);
        cu_stw = ((unsigned long)data[0] << 8) + data[1];

        return cu_stw;
}

/*********************************************************************
 * get_i_digital()
 *
 * @return i_digital  2-byte digital input control word field
```

```
 *
 * Retrieve the digital input control word field from the data sent
 * by the control unit to the DSProfinet IRT controller.
 *
 * The data starts 2 bytes into the CDB (slot).
 ********************************************************************/
unsigned int get_i_digital()
{
        unsigned int i_digital = 0;

        //Since we have a 16-bit RAM but a 32-bit PCI, for every 16
        //bits that get returned there will be 16 bits of 0x000 padding,
        //hence you must create storage for twice as many chars.
        unsigned char data[4]; //we only wany 2 chars.

        epeek_wrapper(0x1000 + read_slot*256 + 2, data, 2);
        i_digital = ((unsigned long)data[0] << 8) + data[1];

        return i_digital;
}

/*******************************************************************
 * get_stw1()
 *
 * @param  motor_num   Which motor to retrieve for
 * @return status1     2-byte status word 1 field
 *
 * Retrieve the status word 1 field from the data sent by the control
 * unit to the DSProfinet IRT controller.  The offset of where this
 * data is in relation to the other data is as follows:
 *
 *      For motor1, the data starts at the 4th byte
 *      For motor2, the data starts at the (4+18)th byte
 *      For motor3, the data starts at the (4+18+18)th byte
 *      For motor4, the data starts at the (4+18+18+18)th byte
 ********************************************************************/
unsigned int get_stw1(int motor_num)
{
        unsigned int status1 = 0;

        //The offset into the read_slot.  So if we are reading CDB
        //3 (the 4th slot in the buffer) then this is how far into
        //the slot to go to get the data.
        int offset = 0;

        //Since we have a 16-bit RAM but a 32-bit PCI, for every 16
        //bits that get returned there will be 16 bits of 0x000 padding,
        //hence you must create storage for twice as many chars.
        unsigned char data[4]; //we only wany 2 chars.

        offset = 4 + (motor_num - 1)*18;

        epeek_wrapper(0x1000 + read_slot*256 + offset, data, 2);
        status1 = ((unsigned long)data[0] << 8) + data[1];

        return status1;
}

/*******************************************************************
 * get_actual_speed()
 *
 * @param  motor_num   Which motor to retrieve for
 * @return speed       4-byte actual speed
 *
 * Retrieve the 32-bit actual speed field from the data sent by the
 * control unit to the DSProfinet IRT controller.  The offset of where
 * this data is in relation to the other data is as follows:
 *
 *      For motor1, the data starts at the 6th byte
```

```
 *      For motor2, the data starts at the (6+18)th byte
 *      For motor3, the data starts at the (6+18+18)th byte
 *      For motor4, the data starts at the (6+18+18+18)th byte
 ****************************************************************/
unsigned long get_actual_speed(int motor_num)
{
        unsigned long speed = 0;

        //The offset into the read_slot.  So if we are reading CDB
        //3 (the 4th slot in the buffer) then this is how far into
        //the slot to go to get the data.
        int offset = 0;

        //Since we have a 16-bit RAM but a 32-bit PCI, for every 16
        //bits that get returned there will be 16 bits of 0x000 padding,
        //hence you must create storage for twice as many chars.
        unsigned char data[8]; //we only wany 4 chars.

        offset = 6 + (motor_num - 1)*18;

        epeek_wrapper(0x1000 + read_slot*256 + offset, data, 4);
        speed = ((unsigned long)data[0] << 24) +
                ((unsigned long)data[1] << 16) +
                ((unsigned long)data[2] <<  8) +
                data[3];

        return speed;
}

/****************************************************************
 * get_stw2()
 *
 * @param  motor_num   Which motor to retrieve for
 * @return status2     2-byte status word 2 field
 *
 * Retrieve the status word 2 field from the data sent by the control
 * unit to the DSProfinet IRT controller.  The offset of where this
 * data is in relation to the other data is as follows:
 *
 *      For motor1, the data starts at the 10th byte
 *      For motor2, the data starts at the (10+18)th byte
 *      For motor3, the data starts at the (10+18+18)th byte
 *      For motor4, the data starts at the (10+18+18+18)th byte
 ****************************************************************/
unsigned int get_stw2(int motor_num)
{
        unsigned int status2 = 0;

        //The offset into the read_slot.  So if we are reading CDB
        //3 (the 4th slot in the buffer) then this is how far into
        //the slot to go to get the data.
        int offset = 0;

        //Since we have a 16-bit RAM but a 32-bit PCI, for every 16
        //bits that get returned there will be 16 bits of 0x000 padding,
        //hence you must create storage for twice as many chars.
        unsigned char data[4]; //we only wany 2 chars.

        offset = 10 + (motor_num - 1)*18;

        epeek_wrapper(0x1000 + read_slot*256 + offset, data, 2);
        status2 = ((unsigned long)data[0] << 8) + data[1];

        return status2;
}

/****************************************************************
 * get_g1_stw()
 *
```

```
 * @param  motor_num   Which motor to retrieve for
 * @return g1stw       2-byte encoder 1 status word field
 *
 * Retrieve the encoder 1 status word field from the data sent by the
 * control unit to the DSProfinet IRT controller.  The offset of where
 * this data is in relation to the other data is as follows:
 *
 *      For motor1, the data starts at the 12th byte
 *      For motor2, the data starts at the (12+18)th byte
 *      For motor3, the data starts at the (12+18+18)th byte
 *      For motor4, the data starts at the (12+18+18+18)th byte
 ****************************************************************/
unsigned int get_g1_stw(int motor_num)
{
        unsigned int g1stw = 0;

        //The offset into the read_slot.  So if we are reading CDB
        //3 (the 4th slot in the buffer) then this is how far into
        //the slot to go to get the data.
        int offset = 0;

        //Since we have a 16-bit RAM but a 32-bit PCI, for every 16
        //bits that get returned there will be 16 bits of 0x000 padding,
        //hence you must create storage for twice as many chars.
        unsigned char data[4]; //we only wany 2 chars.

        offset = 12 + (motor_num - 1)*18;

        epeek_wrapper(0x1000 + read_slot*256 + offset, data, 2);
        g1stw = ((unsigned long)data[0] <<  8) + data[1];

        return g1stw;
}

/****************************************************************
 * get_actual_pos1()
 *
 * @param  motor_num   Which motor to retrieve for
 * @return position1   4-byte encoder 1 actual position value 1 field
 *
 * Retrieve the encoder 1 actual position value 1 field from the data
 * sent by the control unit to the DSProfinet IRT controller.  The
 * offset of where this data is in relation to the other data is as
 * follows:
 *
 *      For motor1, the data starts at the 14th byte
 *      For motor2, the data starts at the (14+18)th byte
 *      For motor3, the data starts at the (14+18+18)th byte
 *      For motor4, the data starts at the (14+18+18+18)th byte
 ****************************************************************/
unsigned long get_actual_pos1(int motor_num)
{
        unsigned long position1 = 0;

        //The offset into the read_slot.  So if we are reading CDB
        //3 (the 4th slot in the buffer) then this is how far into
        //the slot to go to get the data.
        int offset = 0;

        //Since we have a 16-bit RAM but a 32-bit PCI, for every 16
        //bits that get returned there will be 16 bits of 0x000 padding,
        //hence you must create storage for twice as many chars.
        unsigned char data[8]; //we only wany 4 chars.

        offset = 14 + (motor_num - 1)*18;
        epeek_wrapper(0x1000 + read_slot*256 + offset, data, 4);

        //The first byte coming back (out of the 4 bytes) is the most
        //significant one.
```

```
        position1 = ((unsigned long)data[0] << 24) +
                    ((unsigned long)data[1] << 16) +
                    ((unsigned long)data[2] << 8)  +
                    data[3];

        return position1;
}

/*********************************************************************
 * get_actual_pos2()
 *
 * @param  motor_num   Which motor to retrieve for
 * @return position2   4-byte encoder 1 actual position value 2 field
 *
 * Retrieve the encoder 1 actual position value 2 field from the data
 * sent by the control unit to the DSProfinet IRT controller.  The
 * offset of where this data is in relation to the other data is as
 * follows:
 *
 *      For motor1, the data starts at the 18th byte
 *      For motor2, the data starts at the (18+18)th byte
 *      For motor3, the data starts at the (18+18+18)th byte
 *      For motor4, the data starts at the (18+18+18+18)th byte
 *********************************************************************/
unsigned long get_actual_pos2(int motor_num)
{
        unsigned long position2 = 0;

        //The offset into the read_slot.  So if we are reading CDB
        //3 (the 4th slot in the buffer) then this is how far into
        //the slot to go to get the data.
        int offset = 0;

        //Since we have a 16-bit RAM but a 32-bit PCI, for every 16
        //bits that get returned there will be 16 bits of 0x000 padding,
        //hence you must create storage for twice as many chars.
        unsigned char data[8]; //we only wany 4 chars.

        offset = 18 + (motor_num - 1)*18;
        epeek_wrapper(0x1000 + read_slot*256 + offset, data, 4);
        position2 = ((unsigned long)data[0] << 24) +
                    ((unsigned long)data[1] << 16) +
                    ((unsigned long)data[2] <<  8) +
                    data[3];

        return position2;
}
```