Ute Documentation: Quick-Start Software Guide

April 13, 2003

Chapter 1

Introduction

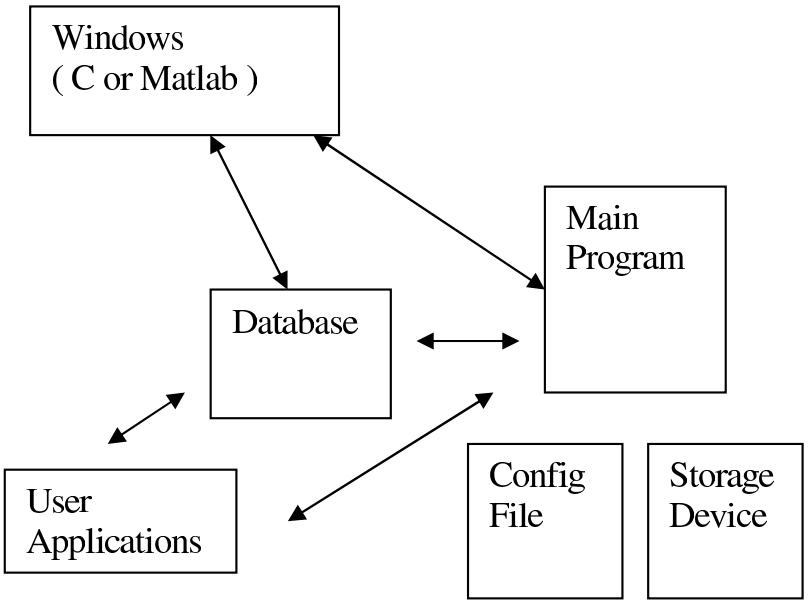

Figure 1.1: A pictorial overview of the ute software architecture.

This manual describes the software architecture for the Holden Utility vehicle built upon the QNX Neutrino real-time operating-system. A high-level view of this architecture and its interactions is shown in Figure 1.1. The central components of the system are the *main program* and common *database*.

The main program provides many server facilities and ...??what else, detects sensors from config file, etc??

TODO: Perhaps "main program" should be renamed "command server" or something more descriptive of its purpose. Also, this introduction is not very clear and should be rewritten and expanded. Each module in Figure 1.1 should be individually described.

TODO: In the future, I would like to greatly expand this introductory chapter to include sections on: the main program, the database, the configuration file, and the Windows-Matlab interface.

The common database acts as a repository for variables to be shared between the main program, the Windows interface, and various user applications. Variables are registered with the database as read-only or read-write. If read-write, then any application (process or thread) may obtain a registered variable and change it, and the variable will automatically be changed in other applications, including the one that originally registered it. It is also possible to trigger a callback function to notify the registering process of any external changes. Information published by the database, such as variable changes, are updated at a fixed frequency. See Section 4.8 for a description of the database API.

A command server (??is this the main program??) is used to register external text commands, so that an application receives a callback function if the user sends a specified message. See Section 4.7 for a description of the command server API.

The functionality of the main program and the database is more-or-less fixed, and new tasks are added to the system as *user applications*. Developing such applications is the focus of this manual.

Presently, this manual consists of two example user applications demonstrating how to obtain information from the configuration file, and to communicate with the database and receive interactive external commands. A description of the architecture *application programming interface* (API) is provided in the final chapter.

1.1 Running UTE.exe

To start the UTE.exe program on a QNX-computer you have to make a telnet connection to QNX. After that you browse yourself into folder that contains "UTE.exe" and configuration file "utez.cfg". To start UTE.exe with configuration file, enter the following command.

```
./ute.exe -cfg:./utez.cfg
```

The program should be now be running.

Chapter 2

Example User Application

This chapter presents two example user applications to demonstrate how a user can write threads, callbacks, and applications to interact with the main program, the database, and other user threads.

Each user application has an entry-point function that is called from the main() function of the main program.[1] This main() function can be found in a file called `main2.c`, where each entry-point function is declared as `extern` (which means "defined in another file"). All entry-point functions have the form:

```
int EntryPointFunc (int *pflag);
```

The `pflag` pointer is used to signal termination to the user application by the main program (user applications should terminate when `*pflag==0`), and the returned integer is 0 for normal termination and non-zero (typically -1) if there is an error.

2.1 An Example: A Basic Application

The following user application is called IniSensor. It is implemented in a file called `sensor.c` and has an entry-point function `int IniSensor(int *pflag)`.

This application demonstrates the construction of a basic sensor-logger: a generic sensor is detected from the configuration file, a new thread is created to handle it, a circular buffer is created to receive the logged data, and a sample-time variable is registered with the common database. Specifically, this example demonstrates the following operations:

- Read information from the configuration file.
- Create and use a circular buffer. This buffer may be read by another user application, such as a "recorder service" for example.
- Print messages to the "printer service"—which sends text information to the console, a file, or across the network.
- Install a periodic task in a new thread.
- Register external text commands, so that the application can receive commands from an external application.

The format of this manual presents the source code of a particular file in short segments (usually one function at a time). Important portions of the code are explained beneath each section using line numbers to identify the relevant code locations.

[1] More precisely, the main() function calls a function IniAll_1() which, in turn, calls each entry-point function. Thus, the writer of a user application must add the new entry-point function to IniAll_1().

2.1.1 Headers and shared variables

The following code segment contains the necessary header files for the command server and data base API, the global (i.e., file scope) variables for this user application, and the function prototypes.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "lib1.h"
5  #include "clx.h"
6  #include "matlablink.h"
7  #include "CmdServer.h"
8  #include "Wvars.h"
9
10 // User defined struct
11 struct myStruct
12 {       int field1,field2 ;
13         float x1 ;
14         unsigned long time ;
15 };
16
17 // Handle to a circular buffer
18 static struct reco *HandleCB1=NULL ;
19 #define LENGHCBUFF 200
20
21 // Callback functions
22 static void ChangeStime(void *a,void *b,void *c);
23 static void *ExeSensor(int *pflag);
24
25 // Initialisation functions
26 int IniSensor(int *pflag);
27 //DynamicLink(IniSensor); // for dynamic linking
28
29 static xuint32 SampleTime=100 ;
30 static int Priority=0 ;
31 static int PrintFlag=0 ;
32
33 static struct matrix matrix1;
34 static xuint16 AINT16_1[100] ;
```

3–8 The header files for the main program and database API library. The functions declared in these headers are described in Chapter 4.

11–15 This structure is used for passing sensor information about via a circular buffer.

18–19 The circular buffer is handled using a `struct reco *` pointer (defined in header file lib1.h). This application stores logged information in a buffer of length `LENGHCBUFF` and other clients will remove the data. It is important that the buffer does not become full, as data will become overwritten.

22–23 Callback functions are used in two cases in this example. The first, `ChangeStime()`, is registered to execute when the application receives an external text command. All callbacks of this type take three `void *` arguments and return nothing (for more information, see below). The second function, `ExeSensor()` is called by the new thread when it is created. Thread callbacks take a single `void *` parameter and return a `void *` parameter.

26–27 The entry-point function `IniSensor()` is called by the main program. Notice that while the local callback functions are declared `static` (i.e., they have file scope), this entry-point function is global. It is also declared `extern` in the main program file main2.c. The dynamic linking function `DynamicLink` is presumably supposed to decouple user applications from compile-time dependency with the main program; it is not presently be used.

29–34 A variety of integer types, such as `xint16` and `xuint32`, are defined in header wvars.h (strangely, they are also defined in header common.h???). Also, there is a matrix type `matrix` defined in wvars.h (see Section 4.2).

2.1.2 Entry-point function

The function `IniSensor()` is the entry-point function for this user application; it is called from the main program. This function searches for the appropriate sensor type in the configuration file and, if present, gets the sensor configuration. It creates a circular buffer for passing logged data, and creates a new thread in which to perform the logging operations. Finally, it registers and external text command for user interaction, and registers some variables with the database.

```
35 int IniSensor(int *pflag)
36 {
37         int st ;
38         int active=0;
39         Printi("INI SENSOR!!\n") ; // send a text message
40
41         // Read some settings from the configuration file
42         GivemeCFG4Entries("SENSOR1","active=[%d],SampleTime=[%d]ms,priority=[%d],
43                 printi=[%d]",&active,&SampleTime,&Priority,&PrintFlag);
44
45         if (active==0){ return(0) ; } // nothing to do.
46
47         SampleTime = LimiteInteger(SampleTime,5,1000) ; // set limits [5,1000]ms
48         Priority   = LimiteInteger(Priority,-1,10)    ; //
49         Priority   = Priority+getprioThread(0); //relative to main thread priority
50
51         // Create circular buffer
52         HandleCB1 = CreateCB("SENSOR1",sizeof(struct myStruct),LENGHCBUFF,0) ;
53         if (HandleCB1==NULL){ return(-1) ; }
54
55         // Create thread to perform job
56         st = CreateThisThread((FunciThread)ExeSensor,pflag) ;
57         if (st<0){     Printi("SENSOR1: can not create thread, bye\n") ; goto bye ; }
58
59         // Register callback for an external command text
60         RegisterExternalCmd("sensor1: change sample time = %d",1,ChangeStime) ;
61
62         // Create matrix
63         IniMatrixStruct(&(matrix1),1,100,MATRIX_INT16,AINT16_1) ;
64
65         // Register variables with database
66         RegistreInteger32Var("SENSOR","SampleTime",&SampleTime);
67         RegistreMatrix("SENSOR","vector1",&(matrix1),&(matrix1.timeStamp));
68
69         return(0) ;
70
71 bye:
72         CloseCBHK(&HandleCB1); // destroy CBuffer
73         Printi("INI SENSOR failed!!\n") ; // send a text message
74         return(-1) ;
75 }
```

39 This print function (and others, see Section 4.1) performs `printf`-like text operations. A normal `printf` is not threadsafe and is too inefficient for use in real-time processes; these functions prevent blocking of time-critical applications by passing messages to a separate printing process.

42–43 The existence of a sensor, and its properties, are defined in the configuration file. The `GivemeCFG#Entry()` functions (see Section 4.6) read a line from the config file and parse parameters from it. For example,

in this code segment, `GivemeCFG4Entry()` finds the config line identified by the string "`SENSOR1`", and extracts four parameters from it. An example config line would be:

```
"SENSOR1: Active=[1],SampleTime=[110]ms,priority=[1],printi=[1]"
```

45 If the sensor is marked as not active, then there is nothing left to do for this application and it exits. Notice, the return value is zero, which implies a non-error return.

47–48 The function `LimiteInteger()` and others (see Section 4.2) ensure that variables fall within specified limits. If they are greater or less than these limits, they are set to the bounding values.

52 The circular buffer here can hold up to `LENGHCBUFF` records of `struct myStruct` objects. If too many objects are added to the buffer, the least recently added objects will be overwritten. The buffer is assigned a logical handle via the string "`SENSOR1`" and the pointer handle `HandleCB1` is used perform actual operations. More information on circular buffer functions is provided in Section 4.3.

56 A new thread is created, and the callback function `ExeSensor()` executes in the new thread (see Section 4.5 for more thread functions). This function is passed `pflag` to permit shutdown signaling. If the thread cannot be created, then the circular buffer is deleted and an error value returned (see lines 71–74).

60 An external text command is registered with the command server (see Section 4.7). In this case, whenever the text command

```
"sensor1: change sample time = xx"
```

is input by the user (where xx is any integer value, say 52), then the callback function `ChangeStime()` is called with `xx` as its argument.

63 A 1×100 matrix variable is allocated.

66–67 Two variables, a 32-bit integer and a matrix, are registered with the database to permit external access. The first string `"SENSOR"` specifies an identifier for a group (or family) of variables, and the second string defines the particular variable tag.

69 Return zero to indicate that the logging thread has been initialised successfully.

2.1.3 Thread execution function

The function `ExeSensor()` is straightforward. It performs the actual sensor-logging work, and operates in its own thread of execution.

```
76 static void* ExeSensor(int *pFlag)
77 {
78         struct myStruct Data ;
79         int Priority=0;
80         int i ;
81
82         PrintiEtc1("sensor, st=[%d]\n",(int)SampleTime) ;
83         setprioThread(0,Priority) ;   // set the desired priority
84         while(*pFlag) // loop untill *pFlag says bye.
85         {
86                 Data.time =(int)GetXTimeNow_10() ; //get current timestamp in [100microsec] units
87
88                 // Do something
89                 Data.x1++ ;
90                 Data.field1=123 ;
91                 Data.field2=Data.field2+11 ;
92                 PutInCBuffer(HandleCB1,&Data); // put last data in the Circular buffer
```

```
93
94              if (PrintFlag) { // send text message to 'printer service'
95                      PrintiEtc1("sensor, t=[%d]\n",(unsigned)Data.time) ;
96              }
97              if (SampleTime<50){ SampleTime=50 ; }
98
99              for (i=0;i<10;i++){ AINT16_1[i]++ ; }
100             matrix1.timeStamp++ ;
101
102             SleepMillisecs(SampleTime); // wait some milliseconds
103     }
104
105     // Loop finishes. Clean up time.
106     CloseCBHK(&HandleCB1); // destroy CBuffer
107     Printi("SENSOR1 task ends\n") ; // say bye
108     return(NULL) ;
109 }
```

84 The thread will execute in this `while` loop until the variable `*pflag` is set to zero by the main program.

86 Various timing functions are described in Section 4.4.

88–92 This section of code is where a real application would perform actual work such as obtaining sensor data and pushing it onto the circular buffer.

94 The `PrintFlag` variable is configured by the configuration file (see line 43).

100 The update of the matrix timestamp simply informs clients who read this variable that it has been updated and contains new information.

102 The value of `SampleTime` determines the loop period of this function. It is assumed that the time required to perform the actual work is much less than the sample period so, at the end of each loop, the thread just sleeps for `SampleTime` milliseconds.

106 When the worker loop eventually terminates (because `*pflag==0`) be sure to delete the circular buffer.

2.1.4 External text command callback function

```
111 static void ChangeStime(void *a,void *b,void *c)
112 {
113     SampleTime = LimiteInteger(*((int*)a),5,1000) ; // some limits [5,1000]ms
114     PrintiEtc1("sensor, new sample time =[%d]\n",(int)SampleTime) ;
115 }
```

111 Notice that `ChangeStime()` takes three `void *` parameters. This is because an external text command callback function can take up to three parameters. In this case, only one parameter is passed: the new sample time.

113 The new `SampleTime` is set.

2.2 A Second Example: GPS Logger

TODO: I have yet to complete the explanations for this section, but it will be similar to the previous section...

2.2.1 Headers and shared variables

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  #include "lib1.h"
5  #include "clx.h"
6  #include "CmdServer.h"
7  #include "Wvars.h"
8
9  // User defined struct
10 struct myStruct {
11         float lati,longi ;
12         xuint32 time ;
13 };
14
15 // Handle for circular buffer
16 static struct reco *HandleCB1=NULL ;
17 #define LENGHCBUFF 200
18
19 // In this example, this task needs the values of some GPS1 variables.
20 static double *pGPS1Lati,*pGPS1Longi ; // pointers to gps1 real position variables
21 static xuint32 *pGPS1Timestamp ; // pointer to timestamp
22
23 static double GPS1Lati,GPS1Longi ; // local copies
24 static float GPS1LatiInMeters,GPS1LongiInMeters ; // local frame GPS position, in meters
25
26 static double GPS1Lati0,GPS1Longi0 ; // reference
27 static float CX=0.0f,CY=0.0f ; // linearization coefficients
28 static xuint32 GPS1Timestamp ;
29
30 static xuint32 FlagSetLocalFrame=0 ;
31 static int active=0;
32 static xuint32 SampleTime=100 ;
33
34 // Callback functions
35 static void ChangeStime(void *a,void *b,void *c);
36 static void ExeOnOff(void *a,void *b,void *c);
37 static void* ExePedrocito(int *pFlag) ;
38
39 // Initialisation functions
40 static int IniExamplePedroStep1(int *pflag);
41 static int IniExamplePedroStep2(int *pflag);
42 static void ExampleHowToReadCfgEntries(void);
```

10–13 Struct for GPS data, which is used to push information onto a circular buffer.

35–37 The first two functions `ChangeStime()` and `ExeOnOff()` are external text command callback functions to change the sampling time and (de)activate sensor logging, respectively. The third callback function is for thread execution.

40–42 Initialisation of the GPS logger is performed in two steps—hence the two initialisation functions. Part of the initialisation process involves reading data from the configuration file; the function `ExampleHowToReadCfgEntries()` gives an example of this.

2.2.2 Entry-point function

This is the entry-point function called from the main program. It is called twice, with a 600ms pause between calls, and is passed `step=1` the first time and `step=2` the second.

```
43 int IniExamplePedro(int step,int *pflag)
44 {
```

```
45          int x ;
46          if (step==1){ x=IniExamplePedroStep1(pflag) ; return(x) ; }
47          if (step==2){ x=IniExamplePedroStep2(pflag) ; return(x) ; }
48          return(-1) ;
49  }
```

46–47 The first time calls the initialisation function on line 46, and the second calls the initialisation function on line 47.

2.2.3 First initialisation function

The first initialisation function reads information from the configuration file, creates a circular buffer, and registers some variables with the database (thereby allowing them to be accessed and altered externally).

```
50  static int IniExamplePedroStep1(int *pflag)
51  {
52          Printi("INI example Pedro, step 1\n") ; // send a text message
53
54          // Read some settings from the configuration file
55          GivemeCFG2Entries("Pedro","active=[%d],SampleTime=[%d]ms",&active,&SampleTime);
56          ExampleHowToReadCfgEntries();
57          if (active==0){ return(0) ; } // nothing to do.
58
59          SampleTime = LimiteInteger(SampleTime,5,1000) ; // set limits [5,1000]ms
60
61          // Create circular buffer
62          HandleCB1 = CreateCB("SENSOR1",sizeof(struct myStruct),LENGHCBUFF,0) ;
63          if (HandleCB1==NULL){ return(-1) ; }
64
65          // Register variables to be published by the DataBAse service
66          RegistreInteger32Var("Pedro","SampleTime",&SampleTime);
67          RegistreInteger32Var("Pedro","SetNewFrame",&FlagSetLocalFrame);
68
69          RegistreFloatVar("Pedro","GPS1LatiM",&GPS1LatiInMeters);
70          RegistreFloatVar("Pedro","GPS1LongiM",&GPS1LongiInMeters);
71          RegistreInteger32Var("Pedro","GPS1TimeStamp",&GPS1Timestamp);
72
73          return(1) ;
74  }
```

55–57 Obtain information from the configuration file. That is, parse the line starting with the identifer `"Pedro"`. Note, the function on line 56 just performs dummy parsing as a demonstration (this function is shown below). If the GPS sensor is configured as not-active, then simply return.

66–71 Several variables are registered with the database under the group identifier `"Pedro"`. Each variable also has an individual identifier string.

73 Notice this initialisation function returns 1 ??why??.

2.2.4 Second initialisation function

The second initialisation function gets pointers to variables stored in the database, registers two external text command callback functions, and starts a new thread for the data-logging task.

```
75  static int IniExamplePedroStep2(int *pflag)
76  {
77          int st ;
78
```

```
79          if (active==0){ return(0) ;}
80          Printi("INI example Pedro, step 2\n") ; // send a text message
81
82          // Ask the database for variables that belong to another owner
83          pGPS1Lati       = GetDoubleDbVarPointer("GPS1_Latitude",TO_ONLY_READ);
84          pGPS1Longi = GetDoubleDbVarPointer("GPS1_Longitude",TO_ONLY_READ);
85          pGPS1Timestamp = GetUInteger32DbVarPointer("GPS1_Timestamp",TO_ONLY_READ);
86
87          // HowToAcceduserDefinedVars(); // alternative method for accessing GPS1 data
88
89          // Check is all the pointers are OK.
90          if ((pGPS1Lati==NULL)||(pGPS1Longi==NULL)||(pGPS1Timestamp==NULL))
91          {       Printi("Pedrocito: BAD, one of the inputs canot be found\n") ; goto BadLuck ; }
92
93          // Register text commands
94          RegisterExternalCmd("chicken pedrocito changes sample time=%d",1,ChangeStime) ;
95          RegisterExternalCmd("pedrocito activity=[%d]",1,ExeOnOff) ;
96
97          // Create thread to do the job
98          st = CreateThisThread((FunciThread)ExePedrocito,pflag) ;
99          if (st<0){      Printi("Pedrocito: can not create thread, bye\n") ; goto BadLuck ; }
100         return(0) ;                                             //OK
101
102 BadLuck:
103         CloseCBHK(&HandleCB1); // destroy CBuffer if it exists
104         Printi("Pedrocito failed!!\n") ; // send a text message
105         return(-1) ;
106 }
```

79 If the GPS sensor was configured as not-active with the first initialisation function, then just return.

83–85 Get pointers to variables from the database. They are marked as read-only so this function cannot alter their state. See Section 4.8 for more information on these database functions.

87 An alternative way to access the GPS data from the database is shown in this function, which appears below.

94–95 Register two external text commands and their associated callback functions.

98–99 Create a new thread to perform the actual data-logging. This thread executes the passed callback function. If the thread cannot be created, delete the circular buffer and return a failure flag (lines 102–105).

2.2.5 Thread execution function

This function performs the actual logging task in its own thread of execution. ...more here...

```
107 static void* ExePedrocito(int *pFlag)
108 {
109         struct myStruct Data ;
110         xuint32 t0,t1 ;
111         xuint32 lastTimeStamp ;
112         int LocalFrameDefinedOk=0 ;
113
114         PrintiEtc1("pedrocito thread running, st=[%d]\n",(int)SampleTime ) ;
115         while(*pFlag) //loop untill *pFlag says bye.
116         {
117                 // If someone cancels task then wait 400 miliisecs and check for change
118                 if (active==0) {
119                         SleepMillisecs(400) ;
120                         continue ;
121                 }
```

```
122
123             SampleTime = LimiteInteger(SampleTime,5,1000) ; // check limits, [5,1000]ms
124             SleepMillisecs(SampleTime); // wait some milliseconds
125
126             // Perform task: Converting GPS data ...
127
128             t0 = *pGPS1Timestamp ;
129             if (lastTimeStamp==t0) { continue ; } // no change in the timeStamp means: nothing new in gps1 data
130
131             GPS1Lati =  *pGPS1Lati ;
132             GPS1Longi =         *pGPS1Longi ;
133             t1 = *pGPS1Timestamp ;
134
135             if (t1!=t0)    {
136                     GPS1Lati =  *pGPS1Lati ;
137                     GPS1Longi =         *pGPS1Longi ;
138                     GPS1Timestamp = *pGPS1Timestamp ;
139             }
140
141             // If someone signals us to set the local frame origin at here, do it now!.
142             if (FlagSetLocalFrame)
143             {       FlagSetLocalFrame=0 ; //reset the flag
144                     GPS1Lati0 =GPS1Lati; // get reference.
145                     GPS1Longi0 =GPS1Longi;
146                     // CX =                    // obtain linearization constants
147                     // CY =
148                     LocalFrameDefinedOk=1 ; // set this flag to say that there is a suitable local frame defined
149                     Printi("pedro: recalculating local frame\n") ; //text message
150             }
151
152             // If exists a local frame definition then obtain relative position
153             if (LocalFrameDefinedOk) {
154                     GPS1LongiInMeters = CX* (float)(GPS1Longi-GPS1Longi0) ;
155                     GPS1LatiInMeters = CY* (float)(GPS1Lati-GPS1Lati0) ;
156
157                     PutInCBuffer(HandleCB1,&Data); // put last data in the circular buffer
158             }
159     }
160
161     // Loop finishes. Clean up time.
162     CloseCBHK(&HandleCB1); // destroy CBuffer
163     Printi("SENSOR1 task ends\n") ; // say bye
164     return(NULL) ;
165 }
```

115 The data-logging thread continues in this `while` loop until `*pflag` is set to zero.

118–121 The `active` flag is configured to be changed by an external text command (see line 95 above and lines 172–176 below). If the user sets the logger to not-active, this thread falls asleep for 400ms and polls the `active` flag again.

128–139 ...

142 The `FlagSetLocalFrame` variable is registered with the database (see line 67) and will be flagged by a different application.

153–158 The GPS data is converted to the ??what sort?? units and pushed onto the circular buffer, for use elsewhere.

162–164 At the end of this function, whereupon the thread terminates, the circular buffer is deleted and this function returns the NULL pointer—signifying normal termination.

2.2.6 External text command callback functions

These two functions are callback functions for external text commands...

```
166 static void ChangeStime(void *a,void *b,void *c)
167 {
168         SampleTime = LimiteInteger(*((int*)a),5,1000) ; // some limits [5,1000]ms
169         PrintiEtc1("pedrocito says: new sample time =[%d]\n",(int)SampleTime) ;
170 }
171
172 static void ExeOnOff(void *a,void *b,void *c)
173 {
174         active = *((int*)a);
175         // Here you can do something more.
176 }
```

2.2.7 Accessing database variables

This function shows an alternative approach to getting information from the database. More information on the database API function on line 183 is given in Section 4.8.

```
177 #include "gps.h"
178 static struct gps *pgpsAll ;
179
180 static void HowToAcceduserDefinedVars(void)
181 {
182   // Call this function during the initialization step
183   pgpsAll = GetUserDbVarPointer("GPS2_AllData",TO_ONLY_READ) ;
184
185   // ... and use any field
186   //    pgpsAll->latitude
187   //    pgpsAll->satellites
188   // etc ...
189 }
```

2.2.8 Reading from the configuration file

The following function demonstrates some examples of using the configuration file API. This API is discussed in Section 4.6.

```
190 static void ExampleHowToReadCfgEntries(void)
191 {
192         char *p ;
193         int integer1=101 ;
194         float float1=1.234 ;
195         float y=33.12 ; int x=11,z=80 ;
196
197         p= GivemeCFGStrEntry("StupidString") ;
198         if (p==NULL) // no cfg entry
199         {       p="I do not know" ; }
200         PrintiEtc1("I think that {%s}\n",p)
201     integer1 = GivemeCFGIntegerEntry("pedroI1",NULL,0) ;
202         float1   = GivemeCFGFloatEntry("PedroF1",NULL,1.23f) ;
203         GivemeCFG3Entries("moreParams","a=[%d],temp is [%f] degrees and life is about [%d] years",&x,&y,&z);
204
205         PrintiEtc3("*** pedro: a=[%d],temp is [%f] degrees and life is about [%d] years\n",x,y,z) ;
206         PrintiEtc3("*** pedro: says=[%s], [%f] [%d]\n",p,float1,integer1) ;
207 }
```

Chapter 3

The Configuration File

This chapter explains how to use the configuration file to initialise different kinds of sensors plugged into the QNX-computer.

3.1 Logger Defaults

```
1 # ———   initial settings ———————
2 #
3 #ExitImmediately [0]
4
5 # ***** recorder default initialization *****
6 DataDirectory:/doc2/Data/
7 DataPrefix:UTE1
8 DataSavingInitialState:[0]
```

1–3 All lines beginning with a `#` are comments.

6–8 The default settings for the data-logging directory, the log-file name prefix, and the whether to start logging immediately the logging program starts, respectively. All these values can be changed on-line.

6 `DataDirectory` is the directory where all log-files are written.

7 The default `DataPrefix` specifies the beginning of the log-file names. For example, the name of the `laser_1` log-file is `UTE1Laser_1.dat`.

8 If the `DataSavingInitialState` flag is one, then the program will automatically begin logging when it starts running. If it is zero, the logging can be commenced by the user at a later time.

3.2 Sensor Configuration

The following code segment identifies the existence of various sensors, and defines their logging configuration. The sensors are three GPSs, two INSs, three lasers, a compass, and an example "virtual" sensor. The sensor "name" at the beginning of each configuration line (e.g., `GPS1`, `GPS2`, `INS1`, `INS2`, `Laser_2`, `Laser_1`, etc) is used as an identifier for that sensor's parameters.

The following parameters are used to configure sensor logging.

- `active` specifies whether a sensor is actually physically connected to the system. If `active=[1]`, then the sensor is enabled, else if `active=[0]` then the sensor is disabled.
- `priority`.

- `com` denotes the com-port number into which the sensor is plugged.
- `speed` specifies the baud-rate of the sensor serial transmission.
- `model` indicates the model number of the sensor. For the Ashtech GPS sensors, there are models 8,12 and 24. For the INS sensors, model 0 is the Watson INS, model 1 is the ??...
- `publish`.

Note, the format of the lines—no white-space, square brackets—is important as data is extracted from this file using a formatting string which must match the format of the appropriate configuration line.

```
9  # **** defined GPSs
10 GPS1 active=[1],priority=[1],com=[7],speed=[4800],model=[8],publish=[1]
11 GPS2 active=[1],priority=[1],com=[4],speed=[9600],model=[24],publish=[1]
12 GPS3 active=[0],priority=[0],com=[5],speed=[4800],model=[12],publish=[1]
13 PPSMode [0] ;
14
15 # **** defined INSs
16 INS1 active=[1],priority=[1],com=[3],speed=[38400],model=[0],publish=[1]
17 INS2 active=[0],priority=[1],com=[5],speed=[38400],model=[0],publish=[1]
18
19 # **** defined LASERSs
20 Laser_2 active=[1],priority=[1],com=[5],speed=[38400],init=[1],publish=[1]
21 Laser_1 active=[1],priority=[1],com=[8],speed=[38400],init=[1],publish=[1]
22 Laser_3 active=[1],priority=[1],com=[9],speed=[38400],init=[1],publish=[1]
23
24 # **** defined COMPASSESs
25 Compass1 active=[0],priority=[1],com=[6],speed=[9600],model=[0],publish=[1]
26
27 # *** virtual sensor, example
28 SENSOR1 active=[1],SampleTime=[100]ms,priority=[1],printi=[0];
```

13 This line specifies whether to install the PPS GPS sensing.

28 ...more here... The virtual sensor introduces its own parameter keywords: `SampleTime` and `printi`. ...

3.3 And the rest... TODO ... complete this

```
29 # ****
30 UTE_ETC active=[1],priority=[1],print=[0]
31 UTE_AI: ai_flags=[11111111],dt=[30]ms,IO_address=[600];
32 prometheus_aqd: IO_address=[500],xx={21};
33
34
35 # ————————————
36 # printer servide settings
37 simulate=[0]
38 ConsolePrintMode [3] ; 0=none, 1=console statistics, 2= console, 3 = UDP
39 ActiveChannelsMask [0000000] ;
40 # ————————————
41 # *** Matlab client service
42 Matlab:active=[1],Dt=[250],ini=[0]
43 # active=[1] -> run thread, active=[1] no.
44 # Dt=[nnn] -> refreshing time,
45 # ini=[1] -> begin running, ini=[0] ->begin paused (can run later, sending it a command)
46 # ————————————
47 # *** UTE listening port for all the internal protocols
48 ListeningIPPort:3210
```

```
# ——————————
# *** matlab client TPC definition
MatlabIPPort:3210
MatlabIP:129.78.210.138
#———————————————
#  *** UDP definition for printer service if ConsolePrintMode:[3]
TextIPPort:3215
TextIP:129.78.210.138


SAVE_CB=[ INS1, GPS1 ]
test_token: AAA, BBB, CCC, DDD, EEE, KKK, LAST

# *** low level controllers, settings
pidA(1):Kp=[0.11],Ki=[0.002],Kd=[0.003],yMax=[11],SeMax=[101]
pidA(2):mode=[1],LeftEndOfRun=[−1000],RightEndOfRun=[1001],manual_output=[1.23],set_point=[12]

pidB(1):Kp=[0.12],Ki=[0.003],Kd=[0.004],yMax=[12],SeMax=[102]
pidB(2):mode=[0],LeftEndOfRun=[−1010],RightEndOfRun=[1011],manual_output=[1.25],set_point=[15]


# *** time syncronization service with external PC video acquisition
VideoSync: active=[0],cx=[400],ini=[1]
VideoPPort: 120
VideoIP: 129.78.210.214
# ——————————————————
# *** DataBAse service setting
DBbaseBaseSampleTime [50]

# ———————————————————
Juan active=[1],SampleTime=[200]ms
# ———————————————————
StupidString YES_THE_DAY_IS_NICE
JuanF1 [ 11.523 ]
JuanI1 [ 155 ]
moreParams a=[321],temp is [5.112]degrees and life is about [81]years
```

Chapter 4

Functions

4.1 Debug

These functions replace `printf()` to perform text message display, primarily for debugging purposes. A standard `printf()` could not be used in a real-time program as it is too inefficient and is not threadsafe. The following functions do not block the real-time tasks, but forward messages to a message-logger process, which performs buffering and periodic print operations. These functions are declared in header clx.h.

Sometimes a message is passed without parameters.

Function	`void Printi(char *str)`
Example	`Printi("INI sensor!!\n");`
Description	Prints a character string.

If variables also form part of the message, then specialisations of `void Printi()` have been defined. For example:

Function	`PrintiEtc1(str,param)`
Example	`PrintiEtc1("New sample time = [%d]\n", (int)time);`
Description	Prints a character string with one formatted parameter. Note: this function is in fact a `#define` so that `param` may be of any type, and the formatting string follows the same conventions as for `printf()`.

Some additional `printf()`-like functions are shown below. Note that the numbers 1, 2, etc indicate the number of formatted parameters the function takes. The use of a macro-based definition prevents the typical ellipsis ($\cdots$) approach used by `printf()`, which parses the formatting string to determine the number of parameters at runtime.

```
#define PrintiEtc2(str,x1,x2) sprintf(buf,str,x1,x2);Printi(buf);
#define PrintiEtc3(str,x1,x2,x3) sprintf(buf,str,x1,x2,x3);Printi(buf);
#define PrintiEtc4(str,x1,x2,x3,x4) sprintf(buf,str,x1,x2,x3,x4);Printi(buf);
```

4.2 Numerical

Several functions are provided to provide upper and lower limits to integer and floating-point variables. These functions are declared in header common.h.

Function	`int LimiteInteger(int x,int lower,int upper)`
Example	`val = LimiteInteger(val, 0, 10);`
Description	If `x` is less than `lower`, `LimiteInteger()` returns `lower`. Else, if `x` is greater than `upper`, `LimiteInteger()` returns `upper`. Otherwise, `LimiteInteger()` returns `x`.

Similar functions are declared in header clx.h (and with better spelling). For example:

```
int limitInt(int x,int lower,int upper)
float limitFloat(float x,float lower,float upper)
```

Various `#defines` are provided for integer types to permit portable control of the integer size. The following types and more are defined in header wvars.h (?? and, it seems, in header common.h—what is going on here??).

```
#define xuint8  unsigned char
#define xint8   signed   char
#define xuint16 unsigned short int
#define xint16  signed short int
#define xuint32 unsigned long int
#define xint32  signed long int
```

A further numerical functionality is the construction of a matrix type. The matrix structure and its creation function are declared in header wvars.h.

```
struct matrix {
    uchar *pmData ;
    xuint32 timeStamp ;
    uint16b nc,nf,nItems ;
    uint32 sz ;
    uchar type,kk ;
};
```

Function	`void IniMatrixStruct(struct matrix *pm,int nc,int nf,int type,void *p)`
Example	`IniMatrixStruct(&myMatrix,10,50,MATRIX_INT16,myData);`
Description	Initialise a matrix type variable. The first parameter is a pointer to the matrix variable itself; the next two are the number of columns and rows, respectively ??check this?? ; the fourth parameter is the matrix type (possible types: `MATRIX_DOUBLE`, `MATRIX_FLOAT`, `MATRIX_INT8`, `MATRIX_INT16`, `MATRIX_INT32`); and the final parameter is a pointer to the matrix data itself. For example, myData would be a 10×50 2-D array: `xuint16 myData [10][50];`. Note, it is critical that `myData` remains in scope while ever the matrix `myMatrix` is in use.

4.3 Circular Buffer

The circular buffer is used to store incoming data in one process or thread, and permit a different process or thread to extract the data. The buffer is fixed size and non-blocking, and if data is added more quickly than it is removed—such that the buffer becomes full—then data will be overwritten. Generally, application writers should ensure the buffer never becomes full. Circular buffer functions declared in lib1.h.

Circular buffer creation and registration.

Function	`struct reco *CreateCB(char *name,int sizeRec,int length,int sizeBlockB)`
Example	`myHandle= CreateCB("MYSENSOR",sizeof(struct myStruct),LENGHCBUFF,0);`
Description	Allocate memory for a circular buffer, assign it a logical handle, and return a pointer handle. The first parameter is a string, which defines a logical handle (or identifier) for the buffer. This enables other applications to obtain a pointer handle to the buffer. The second parameter defines the size of the variable type that will be pushed onto the buffer, and the third sets the maximum number of these objects the buffer will hold before least-recently added data becomes overwritten. The final argument is for alignment ??or something??, and is rarely used—it is usually set to zero. All operations with the buffer are performed via the returned pointer handle.

Function	`struct reco *CreateCBHK(struct reco *rr)`
Example	
Description	?? Not sure?? I think it just registers the logical ID of the buffer with the database to permit external access.

Circular buffer de-registration and deletion.

Function	`void CloseCBHK(struct reco **rr)`
Example	`CloseCBHK(&myHandle);`
Description	Deletes a circular buffer and removes it from the database.

Adding and extracting information from the circular buffer.

Function	`void PutInCBuffer(struct reco *rr,void *prec)`
Example	`PutInCBuffer(myHandle, &myData);`
Description	Add data to the circular buffer. Note, the data information must be of the correct type for the buffer. For example, `myData` is of type `struct myStruct`.

** TODO ** Where is the function for getting data from the CB??

4.4 Time

Timing functions declared in header common.h...

Function	`unsigned long GetXTimeNow_10(void)`
Example	
Description	

Function	`unsigned long GetXTimeNow(void)`
Example	
Description	

4.5 Thread

Sensor logging is typically performed in a separate thread for each sensor. The following API is a wrapper of the POSIX threads API, and provides functions for thread creation, setting and getting thread priorities, and sending a thread to sleep. Thread functions declared in header lib1.h.

Thread creation.

Function	`int CreateThisThread( void*(*Func)(void *), int *pflag)`
Example	`h= CreateThisThread((FunciThread)MyFunc,pflag);`
Description	Creates a thread and executes the user-supplied function in this new thread. It returns 0 if successful, otherwise the thread could not be created. The user-supplied function must take a `void*` parameter and return `void*`. A typedef for this type of function is provided in header common.h: `typedef void*(*FunciThread)(void*);`

Thread priorities.

Function	`void setprioThread(int th,int NewPriority)`
Example	
Description	

Function	`int getprioThread(int th)`
Example	
Description	

Send a thread to sleep.

Function	`void SleepMillisecs(int ms)`
Example	`SleepMillisecs(50);`
Description	Send the current thread to sleep for `ms` milliseconds.

Function	`void SleepSeconds(int sec)`
Example	`SleepSeconds(2);`
Description	Send the current thread to sleep for `sec` seconds.

4.6 Configuration File

Configuration file API declared in header clx.h...

Function	`char *GivemeCFGStrEntry(char *head)`
Example	
Description	

```
float GivemeCFGFloatEntry(char *indicator,int *pflag,float defaultValue);
int GivemeCFGIntegerEntry(char *head,int *pflag,int defaultValue);
```

Function	`int GivemeCFG4Entries(char *head,char *frmt,void *pi1, void *pi2,void *pi3,void *pi4)`
Example	
Description	`"GPS1: Active=[1],SampleTime=[110]ms,priority=[1],printi=[1]"`

4.7 Command Server

The command server API declared in header CmdServer.h.

Function	`void IniCmdServer(int *pflag)`
Example	
Description	

Function	`void RegisterExternalCmd(char *s,int n, void (*Funci)(void *p1,void *p2,void *p3) )`
Example	
Description	

Function	`int RegistreMSGX( unsigned code, void (*funci)(void *p), int sz, int prio)`
Example	
Description	

Function	`int RegisterCmdService(int code, void (*pfun) (void *pdata,uint sz,uint code),int maxSz)`
Example	
Description	

4.8 Database

Database API declared in header wvars.h.

Register variables with database:

Function	`int RegistreVar(char *family,char *name,uchar type,unsigned int size, void (*pfunCall)(void *pdata,uchar icode,int step), void *pvar,uchar icode,int flag, xuint32 *pTimeFlag)`
Example	
Description	

Function	`int RegistreInteger32Var(char *family,char *name,xint32 *pvar)`
Example	
Description	

Function	`int RegistreMatrix(char *family,char *name, struct matrix *pm,xuint32 *ptime)`
Example	
Description	

Obtain variables from database:

Function	`xint32 *GetInteger32DbVarPointer(char *name,int toWhat)`
Example	
Description	

Function	`double *GetDoubleDbVarPointer(char *name,int toWhat)`
Example	
Description	

Function	`void *GetUserDbVarPointer(char *name,int toWhat)`
Example	
Description	