Contents

Test Bed Engine

Chapter 1

1.1 Test Bed Login

The Test Bed Engine (TBE) must be started before structured tests can be performed, or the CAN monitor can be used.

Figure 1.1 shows an overview of the CAN cards installed in the test bed computer (TBC). The bottom part of the figure contains a list of which satellite subsystems are simulated on what CAN port. The users must fill in the name of the subsystems belonging to each port.

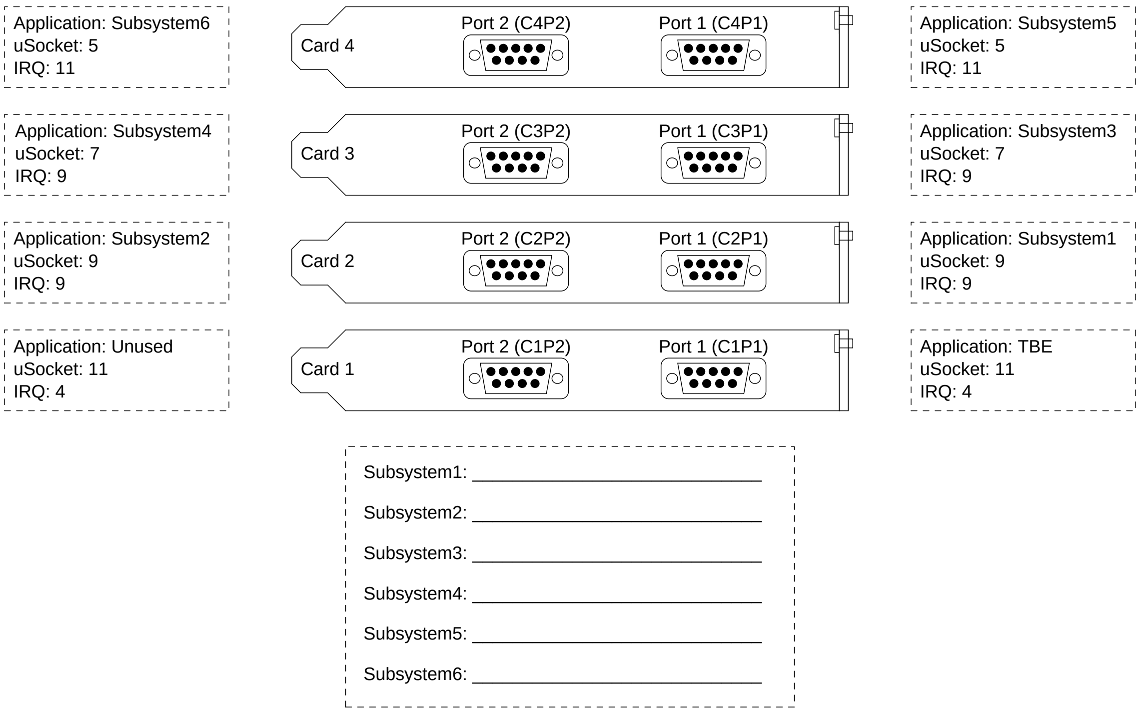


Figure 1.1: *The CAN cards in the test bed computer, as seen from the rear of the computer.*

The TBC is a 2.4 GHz Pentium 4 PC running Mandrake Linux 9.2. The first thing to do when operating the TBC is to log on. The user name and password is:

```
User name: root
Password: testbed
```

When logged in, the next step is to start the graphical environment XFree86. This is done by issuing a `startx` command in the shell following the login prompt.

The `startx` command starts the Windowmaker window manager on top of XFree86. The desktop of Windowmaker is shown in figure 1.2.

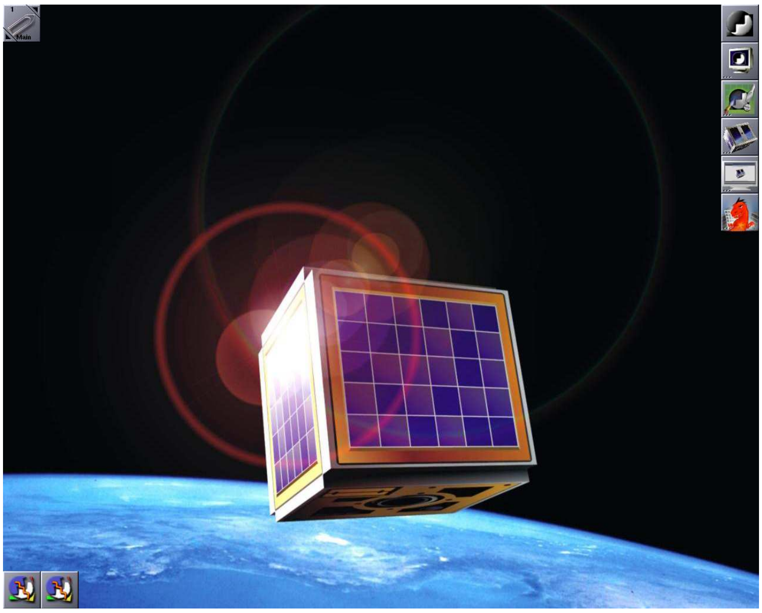


Figure 1.2: *The desktop of the TBC.*

The desktop icons related to the test bed, are the three icons located in the bottom of the right side of the screen. These three icons starts the TBE, the CAN monitor, and the web interface, as described in table 1.1.

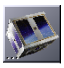
Icon:	Command:	Description:
	`testbed`	Start Test Bed Engine
	`testbedgui`	Start CAN Monitor Unit
	`firebird`	Start Web Interface Unit

Table 1.1: *The icons for controlling the test bed, and the commands executed.*

Note that the command for starting the web interface unit is `firebird`. This command starts a browser, because the actual web interface unit is started when the Apache web-server installed on the TBC is started. This is done automatically when the TBC is started.

1.2 Running TBE

By double clicking on the top most icon, the TBE is started in a terminal. This terminal is shown in figure 1.3.

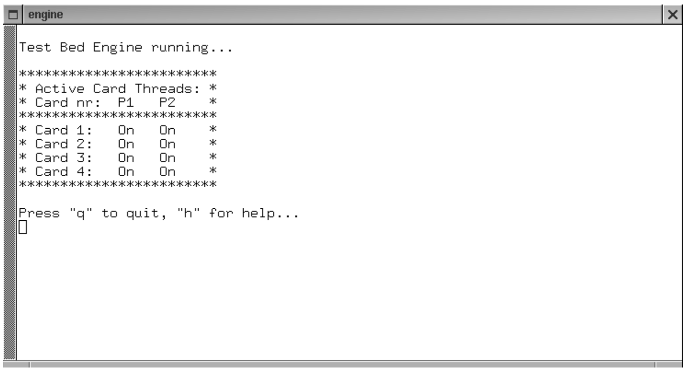


Figure 1.3: *The TBE terminal.*

When this screen is displayed, the TBE is ready to serve the CAN monitor and the web interface.

Before the TBE is started, a number of kernel modules must be loaded. This is done automatically when starting Windowmaker, but the modules can also be managed manually by using the following commands:

Command:	**Description:**
`testbedload`	Loads the modules and creates device files for CAN cards.
`testbedunload`	Unloads the modules.

Table 1.2: *Commands for managing kernel modules needed by the TBE.*

From the TBE terminal in figure 1.3, the TBE can be stopped by pressing "q".

1.2.1 Managing CAN Cards

A help menu can be shown by pressing "h". The help menu is intended for debug purpose, and shown in figure 1.4. The contents of this menu makes it possible to perform actions directly on the CAN cards.

The top most group of keys in the menu are used to select which CAN card the other key groups operate on. The middle group of keys is labelled "Communication Keys", and used to transmit random CAN frames. The last group of keys is used to perform administration tasks on the CAN cards. A technical description of these tasks can be found in the Softing manual.

1.3 Subsystem Simulation

To simulate a subsystem in the TBE software, the subsystem code must be implemented in a specific C file. The TBE is located in /testbed/tbe, and this directory contains a subdirectory for

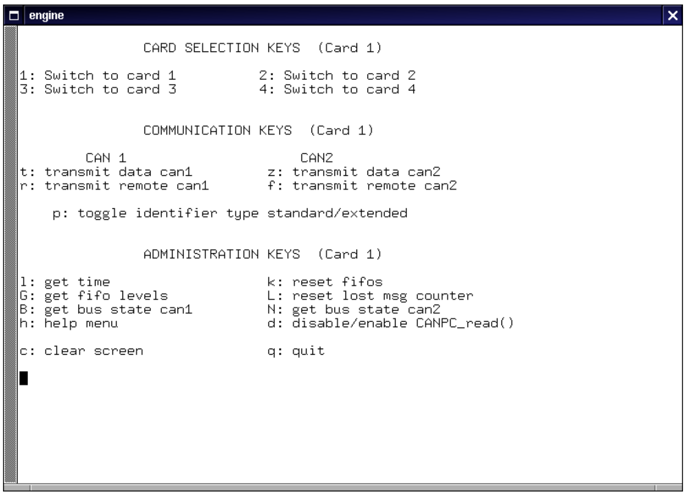


Figure 1.4: *The TBE help screen.*

Subsystem:	**File path:**
Subsystem1	/testbed/tbe/card2/port1.c
Subsystem2	/testbed/tbe/card2/port2.c
Subsystem3	/testbed/tbe/card3/port1.c
Subsystem4	/testbed/tbe/card3/port2.c
Subsystem5	/testbed/tbe/card4/port1.c
Subsystem6	/testbed/tbe/card4/port2.c

Table 1.3: *The subsystems and their file paths.*

every card number. In each of these directories a port1.c and port2.c is present. The subsystems are simulated on card 2, 3, and 4, giving the set of subsystem files listed in table 1.3.

The structure of these files are all identical. Each file contains two functions and a thread which is signalled every time a CAN frame is received. (Note: A subsystem thread is NOT signalled when a planned test is running, IF the subsystem is not specified as part of the test.) The functions and the thread is listed in table 1.4 for subsystem 1.

Function/thread:	**Description:**
C2P1main(void)	Function for creating the thread and configuring the acceptance filter of the port.
C2P1stop(void)	Function for stopping the thread.
*C2P1ThreadFunction(void *arg)	Thread signalled when incoming frames are received.

Table 1.4: *The functions and thread of a subsystem.*

The "C2" in the function names refer to "Card 2" and "P1" refers to "Port 1".

The two functions and the thread of table 1.4 are described in the following.

1.3.1 C2P1Main()

This function is executed when the TBE is started. The function configures the acceptance mask and acceptance code for the packet filtering, done by the CAN cards. CAN frames with identifiers that do not match the bit pattern of the filter are discarded by the CAN cards, without notifying the application or generating interrupt. A filter contains two registers:

Acceptance mask:	Defines which bits to consider in the identifier.
Acceptance code:	Defines the value of the considered bits.

When setting the filter, a bit value of "1" in the mask means that the bit is to be considered, and a "0" means that the bit is a don't care. A "1" in the acceptance code means that IF this bit is to be considered (i.e. it is set in the acceptance mask), then the corresponding identifier bit must have a value of "1" to be accepted. For an acceptance code bit value of "0" the identifier bit has to be "0" as well, taken that the same bit is "1" in the acceptance mask.

Since the test bed can operate on both standard and extended identifiers, two versions of the filter registers exist. The source code of `C2P1Main()` is shown below:

```
int C2P1main(void)
{
  int CardNr = 2;
  /* Acceptance filter Port 1 */
  CardSettings[CardNr].ACCEPT_MASK_1 = 0x0000;
  CardSettings[CardNr].ACCEPT_CODE_1 = 0x0000;
  CardSettings[CardNr].ACCEPT_MASK_XTD_1 = 0x00000000L;
  CardSettings[CardNr].ACCEPT_CODE_XTD_1 = 0x00000000L;

  pthread_create(&C2P1Thread,NULL,(*C2P1ThreadFunction),NULL);
  return 0;
}
```

Note that the `C2P1Main()` only configures the acceptance filters in the C data structure CardSettings[]. The actual acceptance filter configuration is done by the routine that initialises the CAN cards. This routine reads the data structure and stores the settings in the appropriate register on the CAN card. This implies that any change of filter settings done after initialisation is not activated.

After storing the filter settings, the `C2P1Main()` creates a pthread for the routine for receiving CAN frames.

1.3.2 C2P1Stop()

The `C2P1Stop()` function does not do anything, except destroying the thread created by `C2P1Main()`. `C2P1Stop()` is called when the TBE is stopped, and the source code is shown below.

```
int C2P1stop(void)
{
  pthread_cancel(C2P1Thread);
  return 0;
}
```

1.3.3 C2P1ThreadFunction()

The `C2P1ThreadFunction()` is created by `C2P1Main()`, and is a thread that is invoked by a signal, every time a CAN frame that matches the acceptance filter is received on the port.

The source code of the function is shown below.

```
void *C2P1ThreadFunction(void *arg)
{
  CanInFrame thisframe;
  CanOutFrame outframe;
  int CardNr;
  int portNr;
  portNr = 1;
  CardNr = 2;
  while(1)
    {
      pthread_cond_wait(&C2P1Cond,&C2P1Lock);

      while(!empty(&portQueue1[CardNr]))
        {
          LockQueue("P1Queue",CardNr);                 /* Lock the Queue */
          thisframe = dequeue(&portQueue1[CardNr]);    /* Dequeue data from PortQueue1 */
          UnlockQueue("P1Queue",CardNr);               /* Unlock the Queue */

          /* ************************************************************************** *
           * SUBSYSTEM SIMULATION CODE BELOW
           * EXAMPLE SUBSYSTEM:
           * If Identifier 200 is received, identifier 300 is replied with data...
           * ************************************************************************** */

          if(thisframe.Ident == 200)
            {
              outframe.Ident = 300;             /* Set outgoing identifier */
              outframe.XMT_data[0] = 0x02;      /* Outgoing B0 */
              outframe.XMT_data[1] = 0x01;      /* Outgoing B1 */
              outframe.XMT_data[2] = 0x0;
              outframe.XMT_data[3] = 0x0;
              outframe.XMT_data[4] = 0x0;
              outframe.XMT_data[5] = 0x0;
              outframe.XMT_data[6] = 0x0;
              outframe.XMT_data[7] = 0x0;
              outframe.DataLength = 8;          /* Set outgoing data length */
              outframe.Xtd = 1;                 /* Send as extended frame */
              outframe.Rtr = 0;                 /* Send as data frame */
              sendFrame(outframe,portNr,CardNr);
            }
        }
    }
}
```

When the thread is created, the function executes into the `while(1)` loop and stops at the `pthread_cond_wait` call. This is the reception point where the signal of incoming frames is received. In this example, the function operates with one instance of the two data structures `CanInFrame` and `CanOutFrame`. These data structures are used for incoming and outgoing CAN frames respectively. The contents of these structures is described in section 1.4.

When the `C2P1ThreadFunction()` receives a signal, the execution continues. First the queue, from which incoming frames are received, is locked. Then the data is taken from the queue and stored in a local instance of the `CanInFrame` structure. The queue is then unlocked again.

The next thing to be processed is the actual subsystem simulation code. In this example, it is checked if the received frame has an identifier of 200, and if that is the case, a reply with identifier 300 is sent. The function used for sending is `sendFrame(outframe,portNr,CardNr)`.

When the source code of a subsystem is altered, the TBE needs to be recompiled and restarted for the changes to take effect. This is done by issuing the `make` command in the /testbed/tbe directory.

This command compiles each subsystem, the TBE, and a CAN card library separately, and links the o-files together to a single executable. This executable is linked symbolic to the command `testbed`, which is used to start the TBE.

1.4 Data Structures

The data structures needed when programming subsystem simulation are `CanInFrame` and `CanOutFrame`. These structures are shown in table 1.5 and 1.6.

Name:	**Type:**	**Size:**	**Description:**
`Ident`	Unsigned long	4 bytes	CAN identifier.
`DataLength`	Integer	4 bytes	Length of data.
`RCV_data[8]`	Unsigned char	$8 \cdot 1$ byte	Data bytes received.
`UnixTime`	Unsigned long long	8 bytes	Time stamp with resolution of 1 μs.
`frameType`	Integer	4 bytes	The frame type.

Table 1.5: *Parameters in the data structure for incoming CAN frames.*

A description of the possible frame types is given in the Softing manual, in table 4-8.

1.5 Error Handling

The TBE has extensive error handling included. When errors occur, the return value of erroneous functions is evaluated against predefined conditions, and an error handling routine determines whether the TBE should be shut down or continue operation. In both cases the cause of the error is written to a log file testbed.log, placed in /testbed.

Name:	Type:	Size:	Description:	
`Ident`	Unsigned long	4 bytes	CAN identifier.	
`DataLenght`	Integer	4 bytes	Length of data.	
`XMT_data[8]`	Unsigned char	$8 \cdot 1$ byte	Data bytes to be sent.	
`Xtd`	Integer	4 bytes	Extended flag:	1 = Ext. identifier.
				0 = Std. identifier.
`Rtr`	Integer	4 bytes	Remote flag:	1 = Remote frame.
				0 = Data frame.

Table 1.6: *Parameters in the data structure for outgoing CAN frames.*

Web Interface Unit

Chapter 2

The web interface unit of the test bed is used to type in planned tests, run the tests, analyse the test results, and present the result for the user. It is split up into four parts, namely CAN identifiers, test case sets, tests, and test reports. The parts have their own link from the menu on the web-page, as seen on figure 2.1.

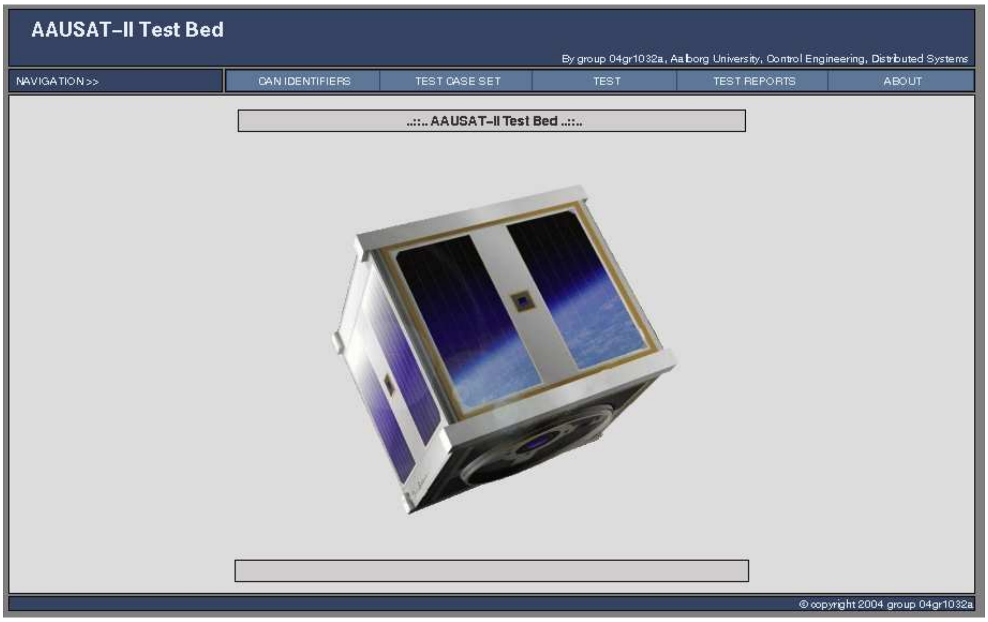


Figure 2.1: *Screen-shot of the first page of the web interface unit.*

The parts are placed chronologically in the order they must be used. The first thing to do, is to type in the CAN identifiers to be used in a test. The next thing to do is to specify the test case sets to be used in the test. After that, the test is specified by a number of user inputs, and a number of test case sets. One test can have one or more test case sets.

When the test is run, a test report is automatically generated, and the result is presented for the user. The CAN identifiers, test case sets, tests, and test reports are all stored in a MySQL database for further use and/or modification.

2.1 CAN Identifiers

When “CAN identifiers” is chosen from the main menu, an overview of all the CAN identifiers is shown. It is possible to order the CAN identifiers by one of the headings. This is done by clicking the heading. Each CAN identifier is edit-able by clicking it. An identifier can also be deleted. If a CAN identifier is deleted, be sure that it is not used in another test. If a new CAN identifier is needed, press the “Create new CAN identifier” link, and the screen shown in figure 2.2 appears.

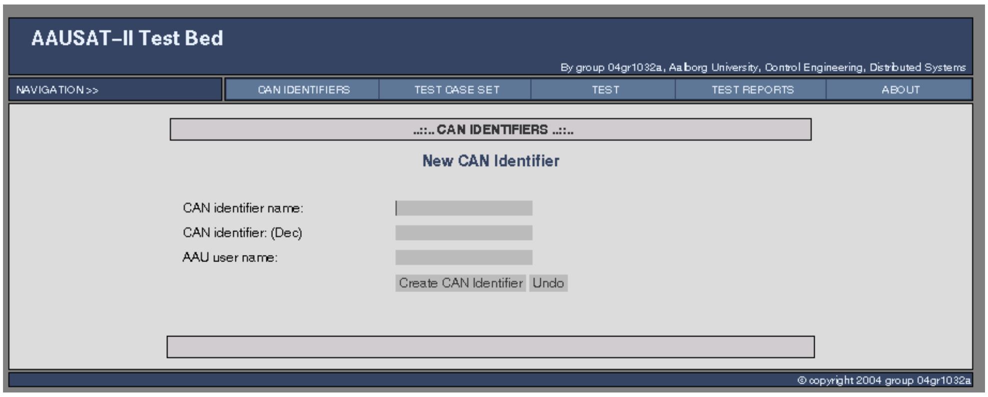


Figure 2.2: *Screen-shot of the user interface to type in CAN identifiers.*

A CAN identifier is given by a name, the identifier (given in decimal format) and the AAU user-name of the author.

Since the communication on-board AAUSAT-II is only CAN frames with extended identifiers, only extended identifiers can be sent by the web interface unit of the testbed.

2.2 Test Case Sets

The test case sets contains the CAN identifiers and the data to be sent on the CAN bus, and the CAN identifiers and data expected to appear on the CAN bus during the test.

When “Test Case Set” is selected from the main menu, an overview of the test case sets is shown. From here it is possible to order the test case sets, delete, view/edit, and create a new test case set.

By clicking the “Create new test case set” link, a screen as figure 2.3 appear. Here the user must decide whether a single frame test case set or an interval test case set is needed. The single frame test sends one frame to the CAN bus, and the interval test sends a number of frames to the CAN bus. For the single frame test, the number of expected outputs must be given by the user, and for the interval test, the number of intervals must be given.

Figure 2.4 shows the user interface of typing in a single frame test case set, where the number of expected outputs is chosen to be four. The interface is split into three groups. The top most group contains the name, a textual description of the test case set, and the AAU user name of the author. In the middle group the input CAN bus frame must be typed in, and in the bottom group, the expected output must be typed in. If many expected outputs have similar data values, and/or CAN identifiers, the copy line can be very useful. The values given in the

Figure 2.3: *Screen-shot of the user interface to type in a new test case set.*

copy line can be copied to all fields below by the "Set" button below each copy field. If all copy fields must be copied to the fields below, use the "Set all" button.

Figure 2.5 shows the user interface of typing in an interval test case set, where the number of intervals is chosen to be two. This interface is split into four groups. The top most group is identical to the single frame test case set. The next group contains a figure to illustrate the borders and test cases to be sent on the CAN bus. The border numbers are shown in top of the figure, and the test case numbers are shown in the bottom of the figure.

The next group contains the fields to define the CAN bus input frames. It is done by giving the intervals of the data fields by typing in the borders. The testbed calculates the test cases to be sent. The bottom group contains the expected outputs on the CAN bus. The test cases in the end of the intervals maybe does not expect a reply. If this is the case, the radio-buttons must be set to don't care for these test cases. The frames are still be sent to the CAN bus to see what happens when a subsystem receives invalid data.

As in the single frame test, a copy line can be used to ease the filling of the forms.

2.3 Tests

The test part contains the test settings for the test to be performed.

When "Test" is selected from the main menu, an overview of the tests is shown. From here it is possible to order the tests, delete, view/edit, and create a new test.

By clicking the "Create new test" button, a screen as figure 2.6 appears. A name and a description must be given to the test, as well as the AAU user name of the test author. The interval time of sending the frames must be given in milliseconds between 1 ms and 10,000 ms, and whether the frames must be sent in random or sequential order must be chosen.

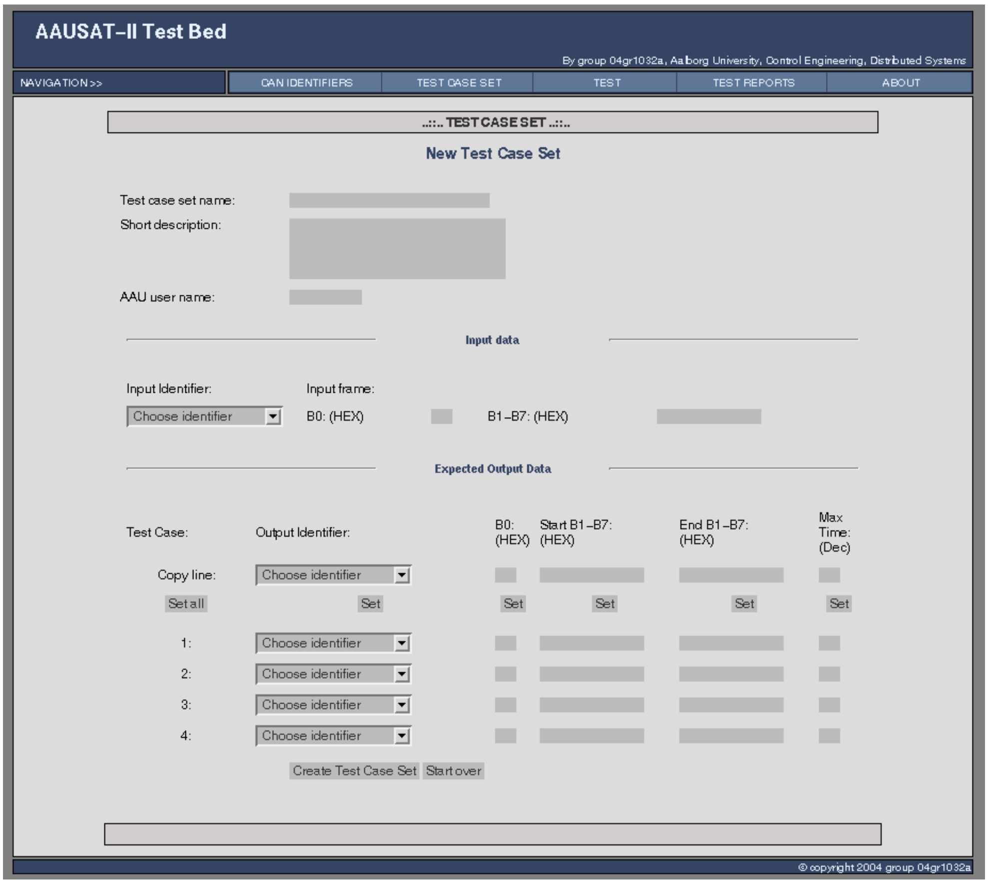


Figure 2.4: *Screen-shot of the user interface for typing in a single frame test case set.*

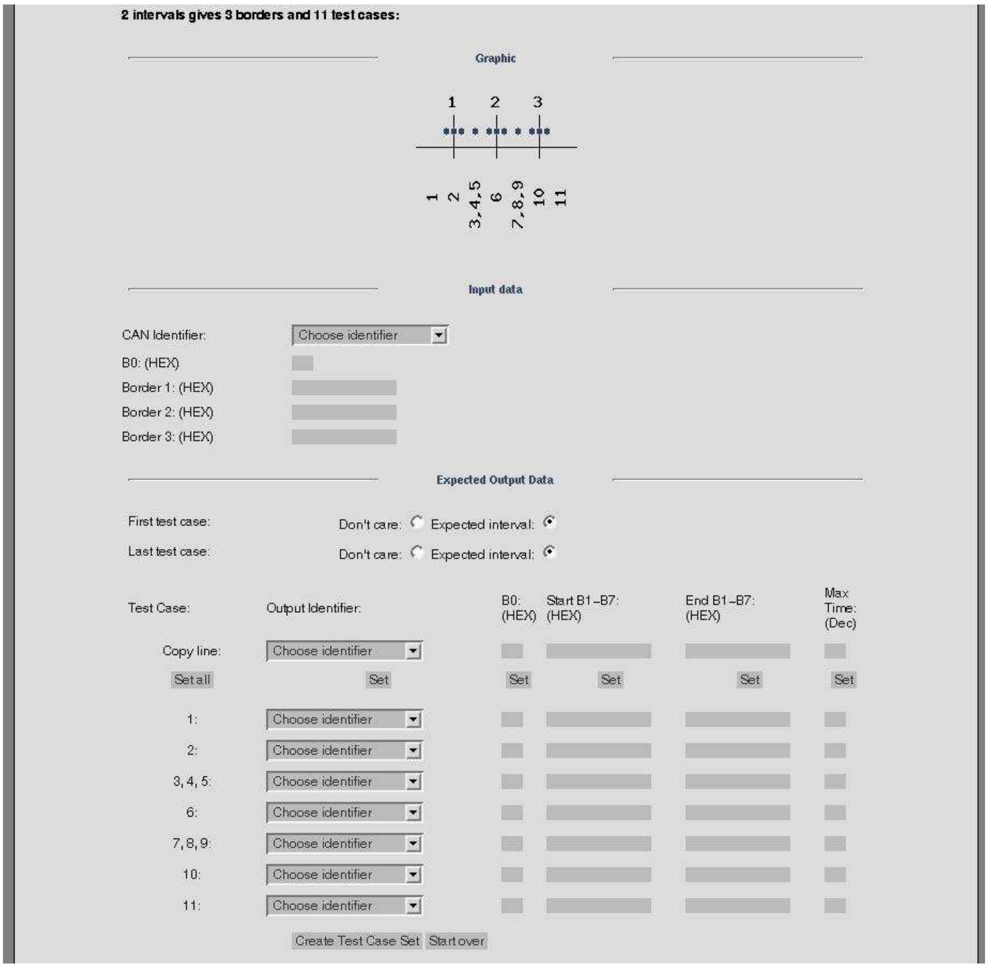


Figure 2.5: *Screen-shot of the user interface for typing in an interval test case set.*

The subsystems can be simulated by using the testbed subsystem drivers. How a subsystem can be simulated can be seen in the testbed engine part of the user manual. The simulated subsystems can be activated/deactivated by the web user interface.

On the right a list of all the test case sets is shown. The test case sets to be included in the test must be selected. It is not possible to start a test without selecting at least one test case set.

The test is started by the pressing the "Create and run test" or "Modify and run test" button. When the button is pressed, the file /testbed/id/test.id is generated. It contains the id of the test to be run. The testbed engine is polling this file, and it starts the test when the file exists. If the file exists, the "run" button in the web user interface will not be present, and a new test cannot be started. The file is deleted by the test bed engine when the test finishes.

In case of computer break down or shut down while running a test, the file might not be deleted, and it is therefore not possible to start a new test. Then the file must be deleted manually.

Figure 2.6: *Screen-shot of the user interface for typing in a new test.*

(Type: rm -f /testbed/id/test.id in a terminal.) If the TBE is shut down manually, the file is removed by the TBE.

When a test is started, a status bar is presented for the user on the web interface. The precise date and time for the test finish is shown, and a counter is counting down the seconds. It is shown in figure 2.7.

2.4 Test Reports

The test report part contains the automatic generated test reports. A test report is generated when a test finishes. The testbed engine uploads the time stamps for the frames sent to the CAN bus to the database, along with all the traffic on the CAN bus. The web interface unit compares the expected frames with the actual frames, and presents the result for the user.

When "Test reports" is selected from the main menu, an overview of the tests reports is shown. From here it is possible to order the test reports, delete, and view a test report.

A test report is split into four parts, as seen on figure 2.8. The top part shows the settings

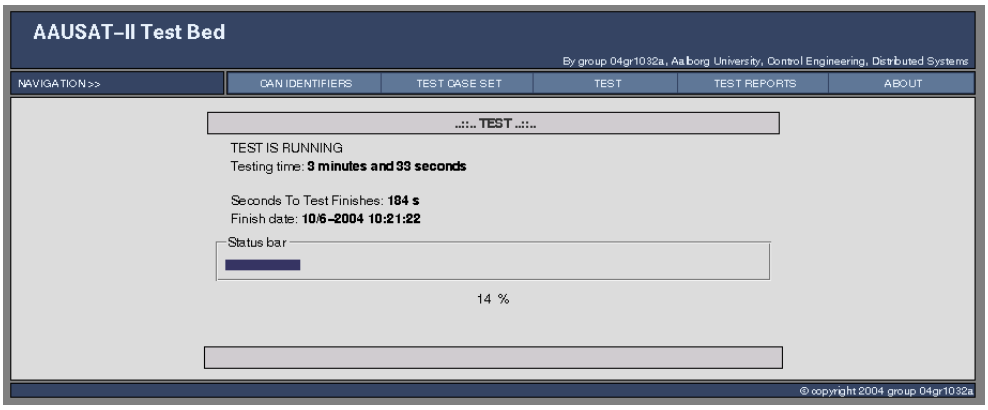


Figure 2.7: *Screen-shot of the test bed web user interface status bar.*

of the test. The next part shows the expected frames and the frame sending time stamps. It also shows whether the expected frames appeared on the CAN bus during the test. A red cross indicates that the expected frame did not appear, and a green check-mark indicates that the frames appeared. If the frame appeared, the frame number of the frame that has validated the expected frame is shown next to the check-mark.

The next part contains all the CAN bus traffic during the test. The frames that has been used to validate expected frames are shown in bold text. In the bottom two .csv files can be downloaded. One contains the expected frames, and one contains the CAN bus traffic. The two files are generated every time the report is shown, if they does not exist already.

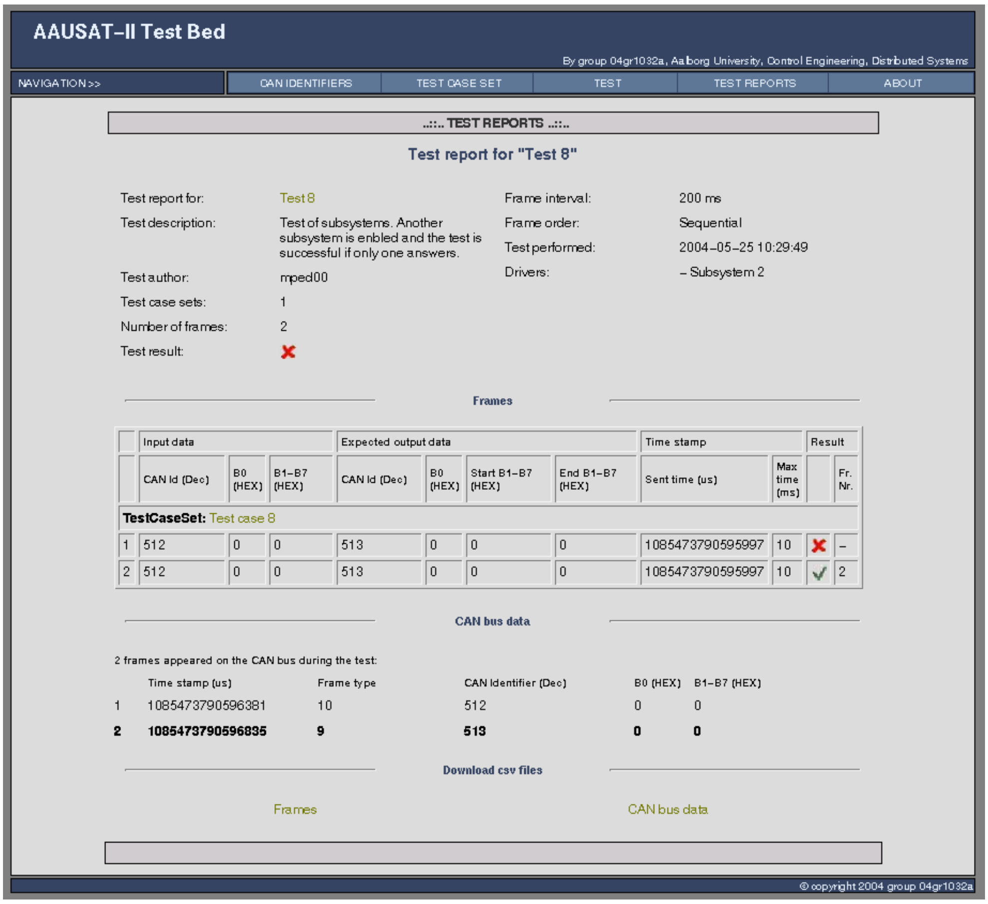


Figure 2.8: *Screen-shot of a test report.*

CAN Monitor Unit

CHAPTER 3

The CAN monitor unit is split into two sections, namely a section which can send CAN frames, and a section which can capture CAN frames from the CAN bus.

The Test Bed CAN Monitor (CMU) is shown in figure 3.1 and the functionality is described in the following.

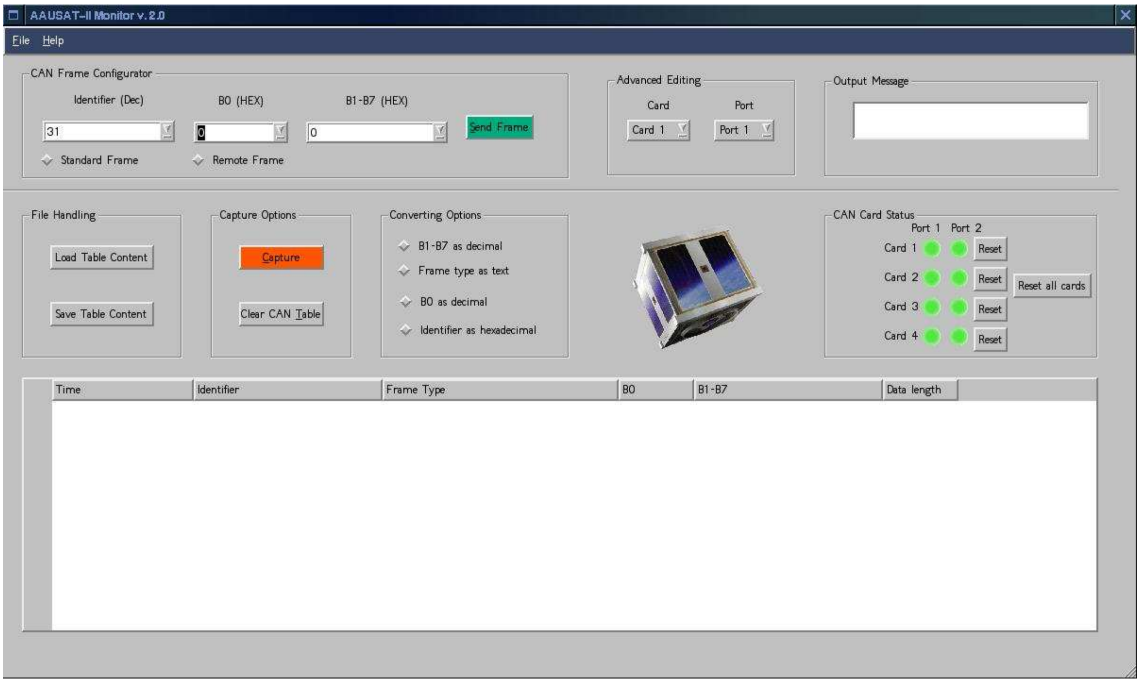


Figure 3.1: *The Test Bed CAN Monitor.*

3.1 CAN Composer

The top part of the CMU, is for composing and transmitting CAN frames to the CAN bus.

The CMU can send standard, extended, and remote CAN frames. These options are selected from radio buttons and drop down menus. Figure 3.2 shows the options for transmitting a frame.

The identifier must be typed in as decimal numbers. If standard frame is selected, the identifier range is between 0-2047, and extended is 0-536870912. The B0 INSANE field is typed in as hexadecimal from 0-FF. The B1-B7 data field is typed in as hexadecimal in the range from 0-FF FFFF FFFF FFFF. The frame can also be transmitted as a remote frame. Then the B0 and

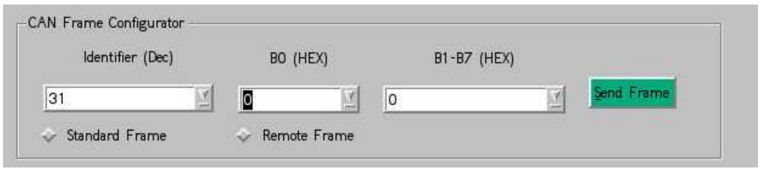


Figure 3.2: *Composing and transmitting a frame.*

B1-B7 does not have any effect. Press the green Send Frame button for sending the composed frame.

The frame can be sent from any of the 8 CAN ports on the Test Bed, this is shown in figure 3.3.

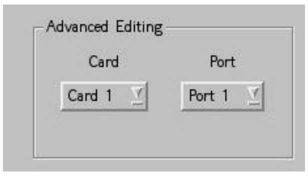


Figure 3.3: *CAN port selection.*

If incorrect data is typed in, the frame can not be sent. A message on the screen displays what is wrong. The last error message is displayed in the output window, shown in figure 3.4. Also feedback information about sent CAN frames is displayed in the output window.

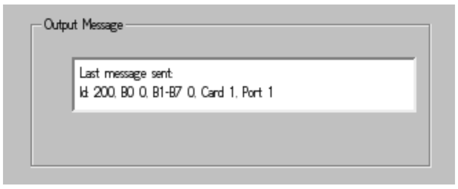


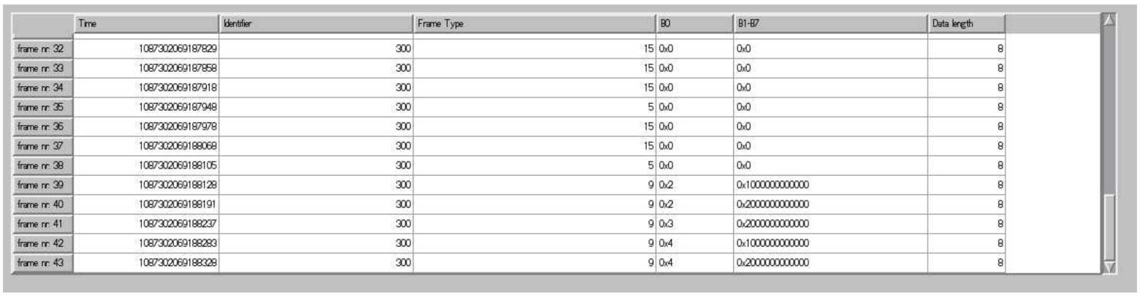
Figure 3.4: *Feedback message.*

3.2 CAN Viewer

The bottom part of figure 3.1 is used for capturing CAN frames from the CAN bus. Every captured frame is shown in a table, as illustrated in figure 3.5 The frame parameters provided in the table are the time of the captured frames (in ms since January 1, 1970), identifier, frame type, INSANE B0, data B1-B7, and data length.

In order to be able to capture CAN frames, the orange capture button has to be enabled. To stop the capturing of CAN frames, press the capture button again. This will disable the CAN bus capture. The capturing button is shown in figure 3.6

The table can be cleaned by pressing the clean button. The capture button has to be disabled.



	Time	Identifier	Frame Type	B0	B1-B7	Data length
frame nr: 32	1087302069187829	300	15	0x0	0x0	8
frame nr: 33	1087302069187858	300	15	0x0	0x0	8
frame nr: 34	1087302069187918	300	15	0x0	0x0	8
frame nr: 35	1087302069187948	300	5	0x0	0x0	8
frame nr: 36	1087302069187978	300	15	0x0	0x0	8
frame nr: 37	1087302069188068	300	15	0x0	0x0	8
frame nr: 38	1087302069188105	300	5	0x0	0x0	8
frame nr: 39	1087302069188128	300	9	0x2	0x1000000000000	8
frame nr: 40	1087302069188191	300	9	0x2	0x2000000000000	8
frame nr: 41	1087302069188237	300	9	0x3	0x2000000000000	8
frame nr: 42	1087302069188283	300	9	0x4	0x1000000000000	8
frame nr: 43	1087302069188328	300	9	0x4	0x2000000000000	8

Figure 3.5: *The table to present the captured frames.*

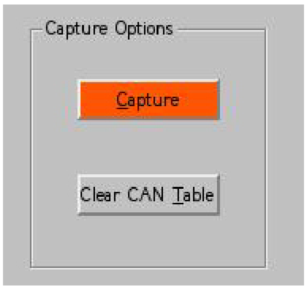


Figure 3.6: *Capture and cleaning frames to the CAN table.*

To store captured CAN frames, disable the capture button and push the save button. A file handling window will appear, where the filename of the file, with the .csv file extension, must be typed in.

To load captured CAN frames, the capture button has to be disabled. Press the load button and select the desired .csv file and press load.

The file handling buttons are shown in figure 3.7.

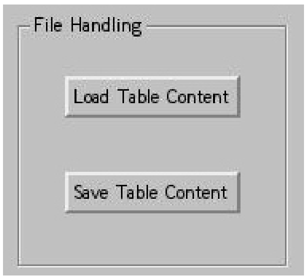


Figure 3.7: *Buttons for file handling.*

To change between the data format of the CAN frames, press the converting radio buttons, for the desired format. This only affects new captured frames. The buttons for changing data format is shown in figure 3.8

If too many error frames are transmitted on the CAN bus, the CAN ports on the CAN cards go into bus off mode. This means that the ports are not able to receive any further CAN frames.

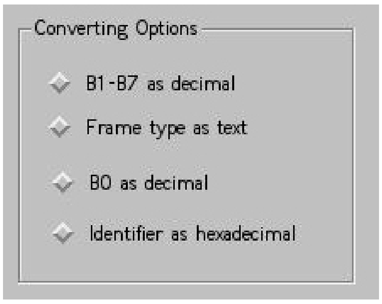


Figure 3.8: *Changing data format for new captured frames.*

The status of each port is indicated by a coloured circle, as shown in figure 3.9.

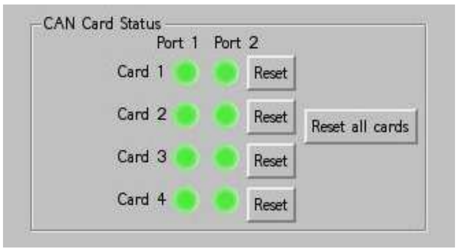


Figure 3.9: *Status of the CAN card ports.*

A green circle means that the given port is OK. A red circle means that the port is in bus off mode, and a yellow circle means that the port is in error passive mode.

When a port has entered bus off mode (red circle), the given can card can be reset by pressing the reset button. If more than one card is going to be reset, the "Reset all cards" button can be used. Only CAN cards in bus off mode are affected of the reset button.

After a reset is carried out, a frame on the CAN bus is transmitted to confirm the reset. The CAN frame contains a standard frame with identifier 2047, B0 = FF and B1-B7 = FFFFFFFFFFFFFF.

Buttons, where a letter in the button label is underlined, can be activated by keypad shortcuts, by pressing ALT + the underlined char.