Manual of the ultracold atom experiment control system "CONTROL"

by Florian Schreck

3/30/2010

Contents

1 Introduction **1**

2 Getting started **3**

2.1 Code navigation . . . 3

2.2 Installation . . . 3

3 Configuring the system **5**

4 Inputs and outputs **10**

4.1 Analog outputs . . . 10

4.2 Digital outputs . . . 11

4.3 DDS outputs . . . 12

4.3.1 Adding a AD9852 DDS . . . 12

4.3.2 AD9852 DDS configuration . . . 15

4.3.3 AD9852 DDS internal frequency ramps . . . 16

4.3.4 Adding a AD9858 DDS . . . 18

4.3.5 AD9858 DDS configuration . . . 18

4.4 Servo motors . . . 18

4.5 Duplicating an output signal . . . 20

4.6 Inputs . . . 21

4.6.1 Digital inputs . . . 21

4.6.2 Analog inputs . . . 21

4.7 Serial and GPIB devices . . . 23

4.7.1 Serial port multiplexer . . . 23

4.7.2 Serial port devices . . . 24

4.7.3 GPIB devices . . . 24

4.7.4 Creating the class describing a serial port or GPIB device . . . 25

4.8 TCP/IP devices . . . 26

4.9 Disabling access to a subsystem . . . 26

5 Parameters **28**

5.1 Adding a parameter . . . 28

5.2 long . . . 29
5.3 double . . . 29
5.4 bool . . . 29
5.5 CString . . . 30

6 Menus **31**
6.1 NewMenu . . . 31
6.2 NewColumn . . . 31
6.3 Static text . . . 31
6.4 Linking experimental sequences to buttons on the user interface . . . 32
6.5 Submenus . . . 33

7 Programming sequences of input/output commands **34**
7.1 Modes of command execution . . . 34
7.2 Programming simple sequences . . . 35
7.3 Timing commands . . . 36
7.3.1 Time reordering . . . 37
7.3.2 Timing accuracy . . . 38
7.4 Storing and retrieving output values . . . 39
7.5 Software waveforms . . . 40
7.6 Serial port device and GPIB device programming . . . 43
7.7 Code example ”Position Servo” . . . 45
7.8 Loops . . . 46
7.9 Idle and WakeUp function . . . 48

8 Main experimental sequence **50**
8.1 Experimental sequence code blocks . . . 50
8.2 Utilities . . . 53
8.3 Triggering the experimental sequence . . . 54
8.4 The DoExperimentalSequence procedure . . . 55
8.5 Communication with the data acquisition program ”Vision” . . . 57

9 User interface **58**
9.1 Set of measurements . . . 59
9.2 Measurement queue . . . 59
9.3 Cyclic operation . . . 59
9.4 The system configuration menu . . . 59

10 Debugging **61**
10.1 Sequence debugging . . . 61
10.1.1 Sequence list . . . 61
10.1.2 MultiIO dump . . . 61
10.2 Hardware debugging . . . 61

10.2.1 Checking address decoding 62
10.2.2 Checking a digital output card 62
10.2.3 Checking the digital inputs 62
10.2.4 Checking an analog input card 62
10.2.5 Checking an analog output card 62
10.2.6 Checking a DDS 62

11 Simple extensions **63**
11.1 Different parallel out card for MultiIO system 63
11.2 Connecting new types of devices to the MultiIO bus system 63

Chapter 1

Introduction

The experiment control system described in this document has been developed with ultracold atom experiments in mind. The design objective is to give the user a simple C programming interface to implement the sequence of a BEC type experiment. The system supports different types of digital and analog output cards, e.g. the National instruments NI653x or NI67xx cards for digital and analog outputs respectively. In addition, analog and digital output cards and direct digital synthesizers (DDS) cards developed by us can be connected to a NI653x digital output card via a bus system. The construction plans for this system can be downloaded from our webpage (http://www.nintaka.com) or obtained by contacting the author (FlorianSchreck@gmail.com). The output speed of this bus system is between 2 million port write commands per second, where one port write command changes the voltage of one single analog output or 16 digital outputs. Any GPIB or serial port command can be issued synchronized with the waveform.

Since Bose-Einstein condensation (BEC) experiments normally require only a limited number of digital and analog inputs and don't require high speed for them, digital inputs are implemented by using the parallel port of the PC and analog inputs are constructed using comparators.

The program communicates with a data acquisition computer, running the data acquisition program "Vision", over TCP/IP. This second computer is responsible to acquire image data and treat and store them together with the experimental parameters. To use the control program it is not necessary to use Vision. Any other type of data acquisition program can be used, or for very simple experiments, Control itself can acquire the data.

Control is constantly being improved. If you consider using it, contact the author to get the latest version of the program (FlorianSchreck@gmail.com).

Chapter 2 **"Getting Started"** explains how to install Control and how to navigate the code.

Chapter 3 **"Configuring the System"** explains how input and output cards are configured.

Chapter 4 **"Inputs and outputs"** explains how inputs and outputs are configured and calibrated. Each input/output obtains a name and a procedure that reads/writes to the input/output. Automatically a user interface is created that allows interactive control of the output. It is explained how new serial port, GPIB port or serial port devices are programmed.

Chapter 5 **"Parameters"** explains how to declare parameters on which experimental sequences depend. Menus for those parameters are automatically generated, the parameters are automatically stored on hard disk, can be stored in and loaded from user specified files, and are transferred to the data acquisition program.

Chapter 6 **"Menus"** explains how to influence the layout of the menus and how to integrate buttons into the menus that call user functions.

Chapter 7 **"Programming sequences of input/output commands"** explains how to create sequences of input/output commands.

Chapter 8 **"Main experimental sequence"** explains how to the main experimental sequence is organized.

Chapter 9 **"User interface"** explains some aspects of the user interface.

Chapter 10 **"Debugging"** explains methods to analyze and debug the experimental sequence and the hardware.

Chapter 11 **"Simple extensions"** explains how to add new output cards.

Chapter 2

Getting started

Knowledge of Visual C++ is not required to use the control program[1]. But it is important to know how to navigate the source code and how to install the program.

2.1 Code navigation

Visual C++ has three tools for code navigation: the "classview", the context menu, and the "search in files" option.

The "classview" is activated with the menu option "View→Classview" and displays all classes of the project. The classes have been arranged in subfolders and the classes you'll most often modify are in the subfolder "Zze most important classes" (the "Zze" guarantees that this folder is displayed last).

The right-mouse click context menu provides "Go to implementation" and "Go to definition" functions.

To navigate the code in Visual C++, the "search in files" option is very useful. It is the fastest way to find the places in the code at which a parameter or a method is used. If you want to find the code which corresponds to a parameter or a method call displayed on the user interface, you simply search for the description displayed on the user interface. If you want to add a new parameter or method, you search for an existing similar parameter or method, duplicate all occurrences and modify the copy. Most code modifications can be done in this way.

Error messages of the control system always explicitly state in which method they were generated (e.g. `COutput::RangeCheck`). To find the location of the initial command that gave rive to trouble, set a breakpoint after the `AfxMessageBox(buf);` command displaying the error message in that method (by left-clicking the gray area next to the code). Confirm the error message. Look through the "call stack" backwards call by call to find the culprit.

2.2 Installation

The most up to date versions of the control program can be downloaded on our website www.nintaka.com. Decompress the file into the directory `C:\SrBEC\Control\` (or similar for other hard disk drives or project names). Open the "control.sln" file using Visual Studio. Find and replace in all files all

[1]If you are interested in this, I recommend the "Scribble" tutorial for Visual C++ available on the Microsoft website http://msdn.microsoft.com.

occurrences of `C:\\` and `D:\\` with the drive letter you want to install the control program to (e.g. `E:\\`). Then search and replace `YourDriveLetter:\\SrBEC\\` (or similar if you start with another than the SrBEC version of control) with `YourDriveLetter:\\YourProjectName\\`. Next, copy the `ControlParam.dat` file to the directory and name given in `CControlApp::CControlApp()` to the variable `ParamFileName`. If you do not have output hardware and still want to play with the program, set `HardwareAccess` to false in `control.cpp`:

```
BOOL CControlApp::InitInstance() {
  ...
  HardwareAccess=false;
  ...
}
```

If you are using "Vision" or our oven control program you have to set the correct IP address of the computer running Vision and the IP port of Vision itself (normally 701) interactively, in the *Configuration parameters* menu and do the same for the oven controller. If you are not using external trigger and clock signals, you need to disable these options in the *Configuration parameters* menu.

When you start control, an error message will most likely be displayed, telling you that the reference parameter file has not been found. Go to the "File" menu, save the parameters into a file of your choice and select this file as new reference parameter file. The current set of parameters is compared to this set of parameters. Changed parameters are labeled red, unchanged green.

Other error messages might be displayed telling you that certain serial port or GPIB port devices have not been found. Simply change the addresses of those devices to `SerialNotConnected` or `GPIBNotConnected` during their registration in `CIOList::CIOList()`; see section 4.7.2 and 4.7.3.

To be able to run Control you need to install traditional NIDAQ from national instruments. The driver for this can be downloaded from the national instruments website.

If your screen has a different resolution from what we are using, you need to change the size of `CEasyDialog` dialog window using the resource editor [2]. Then you need to change the number of dialog elements per column

```
const unsigned int MaxLines=45;
```

in `DialogElement.cpp` to whatever fits your screen best.

[2] The resource editor can be opened using "View→Resource view". Then double click on "DIALOG→CEasydialog" in the resource view.

Chapter 3

Configuring the system

The control software supports national instruments (NI) digital output cards of the types NI6533 and NI6534, analog output cards NI67x3, the input card NI6024E and NI GPIB cards. The parallel port can be used as digital input and with some additional electronic (a comparator) as analog input. A serial port multiplexer allows to attach up to eight devices to the serial port of the PC. Our MultiIO system can be connected to a NI6533 digital output card. It includes self designed analog and digital output boards and direct digital synthesis (DDS) frequency synthesizers.

One and only one NI card of either the NI653x or the NI67x3 type has to be configured as the "master timer" card of the system. It will generate the clock from an internal quartz or an external source and receive the trigger signal. The generation of clock pulses uses the 20MHz oscillator of the NI card, which is divided by an integer constant. Thus you can not reach any arbitrary clock frequency. If you ask for a clock frequency which can not be reached, a warning message is displayed, showing you the next best possible frequency which you should use. A good choice is 2 MHz which corresponds exactly to $0.5\,\mu$s clock period. The NI67xx card is able to transmit the clock pulses through the computer internal flat ribbon cable connecting the different national instruments cards; see NI documentation. This is not possible with the NI6533 card. This means if you have no NI67xx card, but more than one NI6533 card, you should use an external clock. For this you define no card as "MasterTimer" and mark the option external clock in the system parameters menu. It is often beneficial to synchronize the control system to line. For this you need to provide an external clock signal that is synchronized to line and you need to provide a trigger signal in sync with line. Clock and trigger source have to be selected as external in the system parameters menu. See the control system webpage for a design of the required clock and trigger signal generator.

The software needs to be configured to your specific output system configuration. This is done in the routine `CSequence::ConfigureHardware()`. An important task of this routine is to give each output an address, which will be called "software address". To do this, the outputs are added to lists in the order they appear in `CSequence::ConfigureHardware()`. The position in the list is the software address. Three independent lists exist: one for analog outputs, one for digital outputs, and one for DDS. The software addresses are used in `IOList.cpp` to access the output ports. One of the advantages of using software addresses and not directly the hardware addresses of the MultiIO system is that one can easily replace a broken output card with a working one that has a different hardware address. It is just required to change the address of this card in `CSequence::ConfigureHardware()` and the rest of the program remains the same. Another advantage is that e.g. analog outputs of different types (e.g. MultiIO analog outputs and NI card analog outputs) can be addressed in a similar way.

In the following the code contained in `CSequence::ConfigureHardware()` is explained line by line.

```
Serial.RegisterSetSerialPortSubPortFunction(&SetSerialPortSubPort);
```

The `SetSerialPortSubPort()` function is declared in IOList.cpp. The `CSerial` class calls this function to switch the serial port to a specified subport. On www.nintaka.com you'll find a serial port multiplexer circuit. It requires three digital signals to select one of eight serial ports. `SetSerialPortSubPort()` function sets these three digital outputs to the correct values for a specific subport requested by CSerial.

```
Output->SetFrequency(2000000);
```

Sets the bus speed. The unit is Hertz. The maximum speed is 2 MHz, e.g. the value 2000000.

```
Output->SetBufferSize(14*1024*1024);
```

Sets the size of the NI ringbuffer. Unit: 16-bit words. Should be made as big as allowed by the NI driver (and the memory of the computer, which is usually not the limitation). An error message will be displayed if this value is chosen too big.

```
Output->SetLineFrequency(50.0);
```

Line frequency in Hertz (e.g. 50 for Europe or 60 for the US). This information is used for synchronizing commands within the experimental sequence with line (see `SyncToLine` command in Sec. 7.3).

```
Output->SetMaxSyncCommandDelay(0.01);
```

Maximum allowed delay of a GPIB or serial port command in seconds. With windows it is not possible to guarantee perfect synchronization of the serial port and GPIB commands with the experimental sequence. But at least it is possible to determine the timing error. It is usually well below 1 ms. In case it is above the value specified in `SetMaxSyncCommandDelay`, an error message is displayed.

```
GPIB.SetTimeOut(T30s);
```

The maximum amount of time the system waits for GPIB commands to be executed. Look up NI488.h for other possible values besides 30 s, given in the example above (use "search in files" `T30s` to quickly find those definitions).

```
GPIB.SetHardwareAccess(true);
```

GPIB system on (true) or off (false). All GPIB commands will be ignored if the GPIB system is switched off.

```
Output->AddParallelPortAsDigitalInput(0);
Output->AddParallelPortAsDigitalInput(1);
```

Add LPT 0 and LPT 1 as digital input ports.

```
Output->AddNI653x(/*DeviceNr=*/1,/*NrDigitalOutUsed=*/0,/*MasterTimer=*/true,
   /*UseNICardMemory=*/0);
```

Adds a NI6533 or NI6534 card to the system. Add one `Output->AddNI653x` command per card in `CSequence::ConfigureHardware()`. The NI653x cards are labeled from 0 upwards in the sequence in which `AddNI653x` commands appear. The DeviceNr is given by the NI measurement explorer. `NrDigitalOutUsed` ranges from zero to 32 and specifies the number of digital outputs of this card that are directly used for the experiment and not for the MultiIO bus. If this number is zero, all 32 digital outputs can be used for the MultiIO system. If `NrDigitalOutUsed` is 32, then the 32 digital outputs of this card will be added to the list of digital outputs on the system. They obtain software addresses corresponding to their position in the list (e.g. software addresses 0 to 32 if this is the first registration command involving digital outputs). One and only one NI card has to be declared as master timer by setting `MasterTimer` true. `UseNICardMemory` is zero for NI6533 cards and specifies the amount of memory on NI6534 cards.

```
Output->AddNI67x3(/*DeviceNr=*/2,/*NrAnalogOutUsed=*/8,/*DAC12bit*/false,
   /*SlowDigitalOutUsed=*/false,/*SlowDigitalInUsed=*/true,/*MasterTimer=*/true);
   //ana 0..7
```

Adds a NI67x3 card to the system. Add one `Output->AddNI67x3` command per card in `CSequence::ConfigureHardware()`. The NI67x3 cards are labeled from 0 upwards in the sequence in which `AddNI67x3` commands appear. The DeviceNr is given by the NI measurement explorer. `NrAnalogOutUsed` contains the number of analog outputs on the card. This number of outputs are added to the list of analog outputs on the system. They obtain software addresses corresponding to their position in the list. These addresses are noted in the comment after the command. NI67x3 cards contain digital outputs that only be accessed with direct commands (similar to GBIP or serial port commands). If `SlowDigitalOutUsed` is true, these outputs are integrated into the list of digital outputs. It is not recommended to use this option. `SlowDigitalInUsed` configures these digital lines as inputs and places them in the list of digital inputs on the system. One and only one NI card has to be declared as master timer by setting `MasterTimer` true.

```
CNI6024E* AnaIn=Output->AddNI6024E(/*DeviceNr=*/6);
```

Registers a NI6024E input card. The DeviceNr is given by the NI measurement explorer.

```
AnaIn->ConfigureChannel(/*ChannelNr*/0,/*Bipolar*/true,/*Gain0.5,1,10,100*/1);
```

Configures one of the channels of the NI6024E card, see NI6024E manual. You should configure each channel.

```
CMultiIO* MultiIO=new CMultiIO(/*DigBoardNrUsed=*/0);
```

Creates the MultiIO class corresponding the MultiIO system connected to the NI653x card with the software address `DigBoardNrUsed` (this is not the address given by the NI measurement explorer, but the position in the list of NI653x cards as they appear in `CSequence::ConfigureHardware()`).

```
MultiIO->AddDigitalOut(Bus2+14); //dig 0..15
```

Adds a digital output card to the MultiIO system. `BusX` specifies the subbus as set by the dip switches in the subbus decoder bus driver box to which this output card is connected and can range from `Bus0` to `Bus7`; see also MultiIO bus driver circuit hardware manual. The number added to `BusX` is the hardware address of the digital output card as set by the dip switches on the card. This command adds 16 digital outputs to the list of digital outputs on the system. The software addresses correspond to the position of these outputs in the list. As a hint to the user, these software addresses have been placed in the comment after the registration command. One `MultiIO->AddDigitalOut` is required per digital out card. Since our digital output boxes contain two such cards, two commands are needed per digital output box. One address is sufficient for 16 digital outputs since the MultiIO bus data width is 16 bit.

```
for (int j=0;j<8;j++) MultiIO->AddAnalogOut(Bus1+104+j); //ana 8..15
```

Each `MultiIO->AddAnalogOut` command adds an analog output to the MultiIO system. Each analog output card contains eight analog outputs with eight consecutive addresses. Since each analog output uses the full data bus width of 16 bit, each analog output needs an individual address. This is different from digital outputs. There one address is sufficient for 16 outputs. `BusX` specifies the subbus as set by the dip switches in the subbus decoder bus driver box to which this output card is connected and can range from `Bus0` to `Bus7`; see also MultiIO bus driver circuit hardware manual. The number added to `BusX` is the hardware address of the first analog output of the card as set by the dip switches on the card. The three least significant bits of this output are zero. The five dip switches on the card set the five highest bits of the address. Consequently the number given has to be a multiple of eight. The loop variable `j`, ranging from 0 to 7, is added to the address of the first output to step through the addresses of all outputs. The analog outputs are added to the list of analog outputs and obtain software addresses corresponding to the position in the list they obtain. The software addresses are noted in the comment after the command. In case one would like to reverse the order in which the analog outputs are registered the loop has to be changed to `for (int j=7;j>=0;j--)`.

```
MultiIO->AddAnalogOut18Bit(Bus2+64);  //ana 16
```

A 18 bit analog output, which uses four consecutive addresses, is added to the MultiIO bus. The two least significant address lines are used as two additional data lines. These two additional data lines are the two highest significant data lines.

```
unsigned int DDSPPLMultiplier=1;
double InternalClock=300;
double ExternalClock=InternalClock/DDSPPLMultiplier;
```

Some variables that simplify the registration of DDS cards. `DDSPLLMultiplier` is set to the requested multiplier value; see AD9852 datasheet. `InternalClock` is usually 300 MHz. `ExternalClock` is the required external clock frequency.

```
/*AD9852[SrBlueDDSStartNr+0]*/ MultiIO->AddAD9852(/*Bus*/Bus1,/*BaseAddress*/15,
   /*Clockspeed [MHz]*/ExternalClock,/*PLLReferenceMultiplier*/DDSPPLMultiplier,
   /*FrequencyMultiplier*/ 1);
```

Adds an AD9852 direct digital synthesizer (DDS) to the list of DDSs of the system. The software address of the AD9852 DDS is its position in that list. DDSs are mainly used to produce the radio frequency signals required for acousto-optical modulators. You might have several laser systems in your machine, which independently need from time to time an additional AOM. In order to keep the AD9852 DDS registrations corresponding to one laser system grouped together, the software addresses are kept track of by using base addresses for each laser system declared in `IOList.h`. The constant `SrBlueDDSStartNr` in the comment before the registration command is such a base address. To access a AD9852 DDS, commands like `SetFrequencyDDS(SrBlueDDSStartNr+0,Frequency);` are used in `IOList.cpp`. The software address is calculated by `SrBlueDDSStartNr+0` and not directly given. If an additional DDS is added to a laser system, the base addresses of all laser systems registered after the added AD9852 DDS have to be increased by one. Automatically all `SetFrequencyDDS` and similar commands use the new correct software address. `Bus` gives the subbus to which this DDS is connected, as set by the dip switches in the corresponding sub bus decoder box. `BaseAddress` is the base address of this DDS as set by the dip switches on the DDS board. The hardware addresses used by this DDS are `BaseAddress*4+0` to `BaseAddress*4+3`. `Clockspeed` is the clock speed in MHz. `PLLReferenceMultiplier` sets the PLL multiplier value; see AD9852 datasheet. If a frequency multiplier is connected to the DDS, the variable `FrequencyMultiplier` contains the multiplication used (e.g. 2,4,...). If no multiplier is connected this variable is set to 1. The frequency in `SetFrequencyDDS(SrBlueDDSStartNr+0,Frequency);` is the frequency after the frequency multiplier.

```
if ((((SrBlueDDSStartNr+7)+1)!=(SrRedDDSStartNr+0)) || (NrAD9852!=SrRedDDSStartNr))
   AfxMessageBox("CSequence::ConfigureHardware : check DDS software addresses
      in IOList.h");
```

This line of code is there to catch errors in the DDS base addresses of the different laser systems. `SrBlueDDSStartNr+7` is the last used base address. `SrRedDDSStartNr+0` will be the next one used.

```
if (NrDDS<NrAD9852)
  AfxMessageBox("CSequence::ConfigureHardware : increase NrDDS in IOList.h");
```

In case you use more DDS than `NrDDS`, the increase `NrDDS`.

Independent of the AD9852 DDS you can also register AD9858 DDS:

```
/*AD9858[0]*/ MultiIO->AddAD9858(/*Bus*/Bus1,/*BaseAddress*/15,
   /*Clockspeed [MHz]*/ExternalClock,
   /*FrequencyMultiplier*/ 1);
```

This command adds an AD9858 direct digital synthesizer to the list of AD9858 DDSs of the system. The software address of this AD9858 DDS is its position in that list.

```
Output->AddMultiIO(MultiIO);
```

The MultiIO system is added to the output system. In principle there could also be multiple MultiIO systems on different NI6533 cards.

```
Output->Initialize();
```

The output system is initialized.

Chapter 4

Inputs and outputs

After `CSequence::ConfigureHardware()` is executed, lists of digital, analog and DDS outputs and digital inputs have been created. It is possible to access those outputs by their software addresses. But if the experimental sequence would be programmed this way, it would be barely readable. To make it readable, descriptive names like `SetMOTCoilCurrent` have to be given to the outputs. For this, procedures with these names are declared in `IOList.h` and implemented in `IOList.cpp`. A second important function of these procedures is to calibrate the outputs. In the given example, you would like to transmit the current through the MOT coils to `SetMOTCoilCurrent` and not the corresponding voltage required on the analog output. The conversion is done within the `SetMOTCoilCurrent` procedure. These procedures are also registered with the system in the constructor `CIOList::IOList()`. The registration enables many comfortable options. Menus which give manual access to the outputs are automatically created, waveforms can be written to the outputs and the state of the outputs can be accessed. The last step for adding a new output is to initialize it, which is done in `CSequence::InitializeSequence()`.

In the following I'll describe the code modifications required to add an output or input of each type. Very often a similar output exists already on the system. In those cases you let Visual Studio create a list of references to this output by searching for the similar output in all files (e.g. search in all files for `SetZeemanSlowerCurrent`). Then you just need to duplicate and adapt the code Visual Studio did find. The search in all files function is also useful to quickly find the locations in the code I am citing in the following. Just search for the name of the output or input in the SrBEC implementation of Control.

4.1 Analog outputs

The declaration of the output procedure needs to be placed in `IOList.h`.

```
 extern void Set1stZSCurrentPSH(double Current);
```

I also use `Set` as the first three letters of the name of an analog output. I prefer descriptive names, even if they get long. If the name consists of several words, they are directly concatenated and the first letter of each word is written in capital letters.

The implementation has to be placed in `IOList.cpp`.

```
void Set1stZSCurrentPSH(double Current) {
    double Voltage=Current*10.0/200.0;
    Output->AnalogOutScaled(/*software address*/2,/*unscaled value*/Current,
      /*scaled value*/Voltage);
 }
```

Any scaling of output values can be placed here. `Output->AnalogOutScaled` requires the software address of the output you want to use. If an output is used twice, a warning message will be displayed when the program is started. You can also put the value `NotConnected` here, in which case commands to this output are ignored. This is useful if you temporarily modify code or add future options. It is also possible to declare virtual analog outputs that have no corresponding real hardware output. The range of software addresses for such outputs is `HelpAnalogOutStartNumber` to `HelpAnalogOutStartNumber+999`. Examples using virtual outputs are given in sections 7.6 and 7.7.

The output needs to be registered in the constructor of `CIOList`, which is in `IOList.cpp`. The position at which the registration command appears in the constructor determines the position at which the output is displayed in the manual operation menus.

```
RegisterAnalogOutput(&Set1stZSCurrentPSH,"Set1stZSCurrentPSH",
   "Set 1st ZS Coil Current","A","Some help text",/*Min*/0,/*Max*/100,
   /*Constant*/false,/*Color*/ColorRed);
```

`&Set1stZSCurrentPSH` is a pointer to the `Set1stZSCurrentPSH` procedure. The string "`Set1stZSCurrentPSH`" contains the exact same name again, this time as string. It simplifies access to the output and makes it possible to display the name used in the code for this output also on the user interface or elsewhere. "`Set MOT Coil Current`" is a description of the output that is displayed in the manual operation menus. "`A`" is a string displayed after the value in the manual operation menus, usually the units of the output.

All parameters after the units are optional. Some help text"" is a help text which will be displayed if the mouse is hovering above the output value for some while or if the context box to the left of the output is clicked. `Min` and `Max` contain the minimum and maximum value which may be written to this output. If the value requested lays outside this interval, it will be forced on the nearest boundary of the interval. If `Constant` is true, then the user can not manually modify this output. `Color` defines the color of the text displayed in the manual operations menu. Color variables are declared in `color.h` and defined in `color.cpp`. You can add more colors there as you need them.

Another important step for adding an output is to initialize it. This is done by calling the output procedure with the initial value of the output as parameter in `CSequence::InitializeSystem()` for example as follows.

```
    Set1stZSCurrentPSH(Initial1stZSCoilCurrent);
```

`Initial1stZSCoilCurrent` is a parameter that has to be declared, defined and registered in `param.h` and `param.cpp` as explained in chapter 5. Instead of a parameter you could also give a fixed initial value `Set1stZSCurrentPSH(0);`, but this is only useful in cases were the output is definitely completely off or on at the beginning of the experimental sequence. It might also be that you would like to initialize an output only once, when you start the program. In that case you have to place the initialization call in `CSequence::InitializeSystemFirstTime()`.

4.2 Digital outputs

Adding a digital output follows essentially in the same way as adding an analog output.

The declaration of the output procedure needs to be placed in `IOList.h`.

```
extern void SwitchFeshbachPSCIGBT(bool OnOff);
```

In my naming convention, names of digital outputs always start with `Switch`.

The implementation has to be placed in `IOList.cpp`.

```
void SwitchFeshbachPSCIGBT(bool OnOff) {
    Output->DigitalOutScaled(/*software address*/2,OnOff,!OnOff);
 }
```

The only scaling possible for a digital output is inversion, implemented by the exclamation mark in front of the `OnOff` variable. You can also use the value `NotConnected` as software address, in which case commands to this output are ignored. It is also possible to declare virtual digital outputs that have no corresponding real hardware output. The range of software addresses for such outputs is `HelpDigitalOutStartNumber` to `HelpDigitalOutStartNumber+999`. Examples using virtual outputs are given in sections 7.6 and 7.7.

The output needs to be registered in the constructor of `CIOList`, which is in `IOList.cpp`. The position at which the registration command appears in the constructor determines the position at which the output is displayed in the manual operation menus.

```
RegisterDigitalOutput(&SwitchFeshbachPSCIGBT,"SwitchFeshbachPSCIGBT",
    "MOT Light","Some help text",/*Constant*/false,/*Color*/ColorRed);
```

The parameters given to `RegisterDigitalOutput` are similar to the ones given to `RegisterAnalogOutput` and do not need to be explained again.

Again you need to initialize the output for example in `CSequence::InitializeSequence()`:

```
SwitchFeshbachPSCIGBT(On);
```

To simplify the code, `On` and `Off` are defined as

```
const bool On=true;
const bool Off=false;
```

The MOT light can be switched on by `SwitchFeshbachPSCIGBT(On)` and off by `SwitchFeshbachPSCIGBT(Off)`.

4.3 DDS outputs

Most DDS need only to control the frequency and intensity of the signal produced without using the other options provided by the DDS chip as for example DDS internal frequency ramps. We will now first discuss how to implement the code required for frequency and intensity control and later we will describe the additional commands required for DDS internal frequency ramps and other special features of the DDS.

4.3.1 Adding a AD9852 DDS

The declaration of DDS frequency and intensity commands is placed in `IOList.h`.

```
extern void SetFrequencySrBlueMOTSPAOM(double Frequency);
extern void SetIntensitySrBlueMOTSPAOM(double Intensity);
```

The units of the frequency is MHz. The intensity ranges from 0% to 100%. Instead of the `SetIntensity...` procedure it is also possible to declare a `SetAttenuation...` procedure. In that case the radio frequency (rf) intensity is given relative to the maximum rf intensity. The possible values are `DDSAttenuationMax`=-42 dB to 0 dB. Controlling the intensity by setting an attenuation in dB is beneficial for dipole traps. Linear ramps in the attenuation given in dB translate into exponential rf power ramps as needed for evaporative cooling. In addition it is more convenient when dealing with rf power levels with orders of magnitude difference.

```
extern void SetAttenuationSrBlueMOTSPAOM(double Attenuation);
```

The definition of the frequency and intensity commands is placed in `IOList.cpp`.

```
void SetFrequencySrBlueMOTSPAOM(double Frequency) {
  Output->RangeCheck(Frequency,/*Min*/60,/*Max*/80,"SetFrequencySrBlueMOTSPAOM");
  SetFrequencyDDSAD9852(SrBlueDDSStartNr+4,Frequency);
}
void SetIntensitySrBlueMOTSPAOM(double Intensity) {
  SetIntensityDDSAD9852(SrBlueDDSStartNr+4,Intensity);
}
```

Instead of the intensity procedure, the attenuation procedure can be specified.

```
void SetAttenuationKDipoleTrapSPAOM(double Attenuation) {
SetAttenuationDDSAD9852(SrBlueDDSStartNr+4,Attenuation);
}
```

The software address of the DDS is given relative to the first DDS of this laser system, here `SrBlueDDSStartNr`. See also the explanation of this in the discussion of the `CSequence::ConfigureHardware()` procedure in chapter 3.

The output power of an DDS can be adapted in dependence of the frequency of the DDS. This can for example be used to obtain always the same laser power after a double pass AOM independent of the AOM's frequency (within a reasonable range). The calibration of the DDS is contained in a calibration file, which contains the support points of a stepwise linear calibration curve. To activate intensity calibration you have to define and register a virtual digital output that serves to switch the intensity calibration on and off. Here is an example. Place

```
extern void SwitchIntensityCalibrationSrBlueSpectroscopyDPAOM1(bool OnOff);
```

in `IOList.h`,

```
RegisterDigitalOutput(&SwitchIntensityCalibrationSrBlueSpectroscopyDPAOM1,
  "SwitchIntensityCalibrationSrBlueSpectroscopyDPAOM1",
  "Intensity Calibration Spectroscopy AOM 1");
```

in `CIOList::CIOList`

```
void SwitchIntensityCalibrationSrBlueSpectroscopyDPAOM1(bool OnOff) {
    SwitchIntensityCalibrationDDSAD9852(SrBlueDDSStartNr+0,OnOff,CAL_INTENSITY);
}
```

in `IOList.cpp`, and

```
SwitchIntensityCalibrationSrBlueSpectroscopyDPAOM1(On);
```

in `CSequence::InitializeSystem()`. `CAL_INTENSITY` specifies that this DDS is controlled in intensity by a `SetIntensity` command. Alternatively `CAL_ATTENUATION` can be used for a DDS that uses the `SetAttenuation` command for intensity control. The first time `SwitchIntensityCalibrationSrBlueSpectroscopy` is executed a calibration file is loaded. The calibration file name is given in the `LoadDDSCalibration` procedure in `IOList.cpp` (e.g. `d:\SrBEC\ControlSrBEC\Calibrations\DDSCalibrationX.dat`). The `X` in this filename has to be replaced by the software address of the DDS for which this calibration file is foreseen. The format of the calibration file is the following.

```
StartFrequency
StopFrequency
NumberOfCalibrationPoints
Intensity reduction factor at start frequency
...
additional intensity reduction factors
in total NumberOfCalibrationPoints factors needed.
...
Intensity reduction factor at stop frequency
```

If the `SetIntensity` command is used, the "intensity reduction factor" is a number between zero and one that is multiplied with the intensity at which the DDS should be operated. If the `SetAttenuation` command is used, the "intensity reduction factor" is given in dB and added to the attenuation requested. An example to be used with `SetAttenuation` is

```
60
90
5
0
-3
-6
-3
0
```

The `Output->RangeCheck` call verifies if the frequency requested is in the allowed range between `Min` and `Max`. If it is not, an error message is displayed and the frequency is set to $(Min + Max)/2$.

The DDS output procedures need to be registered in the system. This is done in the constructor of `CIOList` as follows.

```
RegisterAnalogOutput(&SetFrequencySrBlueMOTSPAOM,"SetFrequencySrBlueMOTSPAOM",
  "Sr blue MOT (SP) Frequency","MHz","Some help text",/*Min*/60,/*Max*/80,
  /*Constant*/false,/*Color*/ColorRed);
RegisterAnalogOutput(&SetIntensitySrBlueMOTSPAOM,"SetIntensitySrBlueMOTSPAOM",
  "Sr blue MOT (SP) Intensity","0..100%","Some help text",/*Min*/0,/*Max*/100,
  /*Constant*/false,/*Color*/ColorRed);
RegisterAnalogOutput(&SetAttenuationKMOTSPAOM, "SetAttenuationSrBlueMOTSPAOM",
  "Sr blue MOT (SP) Attenuation",DDSUnits,"Some help text",/*Min*/DDSAttenuationMax,/*Max*/0,
  /*Constant*/false,/*Color*/ColorRed);
```

Either the `SetIntensitySrBlueMOTSPAOM` procedure should be registered or the `SetAttenuationSrBlueMOTSPAOM` procedure, not both. The parameters of the `RegisterAnalogOutput` procedure are described in the 4.1 section. For convenience

```
DDSAttenuationMax=-42;
```

```
CString DDSUnits;
DDSUnits.Format("%.0f..0dB",DDSAttenuationMax);
```

were introduced at the beginning of the `CIOList` constructor.

Each DDS has to be initialized. Initial frequency and intensity are set in `CSequence::InitializeSequence`

```
SetFrequencySrBlueMOTSPAOM(SrBlueMOTSPAOMFrequency);
SetIntensitySrBlueMOTSPAOM(SrBlueMOTSPAOMIntensity);
SetAttenuationSrBlueMOTSPAOM(SrBlueMOTSPAOMAttenuation);
```

Also here, either the `Intensity` or the `Attenuation` command have to be used not both. The parameters are declared as described in section **??**.

4.3.2 AD9852 DDS configuration

DDS are microcontrollers with many options. You can access every option by directly accessing the DDS using commands provided by the `CAD9852` class. If you are not using the waveform generation mode, you can access these commands like this:

```
AD9852[KDDSStartNr+4]->InitPLL();
WriteMultiIOBus(/*DebugMultiIO*/false,"D:\\SrBEC\\DebugMultiIO.dat");
```

`AD9852[]` is a table containing pointers to instances of the `CAD9852` class. It is indexed by the software address of the DDS. All possible commands can be found by looking up the `CAD9852` class. The explanation of the commands is given in the AD9852 datasheet, available on the analog devices webpage.
The

```
WriteMultiIOBus(/*DebugMultiIO*/false,"D:\\SrBEC\\DebugMultiIO.dat");
```

command writes all commands that have been issued out on the MultiIO bus. If `DebugMultiIO` is true, the NI card bus buffer is stored in a human readable form as a file with the given filename.

DDS need to be configured before usage. This is done in the `CSequence::InitializeSystemFirstTime` procedure as follows.

```
for (unsigned int i=0;i<NrAD9852;i++) {
  AD9852[i]->SetFrequencyWritePrecision(DDSWritePrecision);
  AD9852[i]->SetComparatorPowerDown(true);
  AD9852[i]->SetBypassInverseSinc(true);
  AD9852[i]->SetExternalUpdateClock(true);
  AD9852[i]->SetControlDACPowerDown(true);
  AD9852[i]->InitPLL();
}
WriteMultiIOBus();
```

The loop goes over the software addresses of all installed DDS. Usually you do not need to change this initialization. The frequency precision is set by `SetFrequencyWritePrecision` to `DDSWritePrecision` bytes. `DDSWritePrecision` is usually three bytes, which allows on a 2 MHz bus to write a frequency command in 1.5 μs. The frequency has then a 8 Hz resolution. Other possible values are

```
//Precision x corresponds to frequency precision y on 150 MHz signal
```

```
// x = y = 150MHz/(256^x)
 1 = 0.5 MHz
 2 = 2.2 kHz
 3 = 8 Hz
 4 = 0.03 Hz
 5 = 0.13 mHz
 6 = 0.5 microHz
```

4.3.3 AD9852 DDS internal frequency ramps

It is possible to perform DDS internal frequency ramps. You can either start such a ramp by direct commands to the DDS as explained in section 4.3.5. But often you need to start such a ramp during the experimental sequence, which means in the ”waveform generation mode”; see section 7.1. For this more analog and digital outputs associated with the DDS have to be declared. In `IOList.h` you have to declare:

```
extern void SetStartFrequencySrRedMOTSPAOM1(double Frequency);
extern void SetIntensitySrRedMOTSPAOM1(double Intensity);
extern void SetStopFrequencySrRedMOTSPAOM1(double Frequency);
extern void SetModulationFrequencySrRedMOTSPAOM1(double Frequency);
extern void SetClearACC1SrRedMOTSPAOM1(bool OnOff);
extern void SetTriangleBitSrRedMOTSPAOM1(bool OnOff);
extern void SetFSKBitSrRedMOTSPAOM1(bool OnOff);
extern void SetModeSrRedMOTSPAOM1(double Mode);
extern void SetRampRateClockSrRedMOTSPAOM1(double Rate);
```

In `IOList.cpp` you have to define:

```
void SetStartFrequencySrRedMOTSPAOM1(double Frequency) {
  DDSAutomaticAttenuationCalibrationMode=CAL_NONE;
  Output->RangeCheck(Frequency,60,100,"SetStartFrequencySrRedMOTSPAOM1");
  SetFrequencyDDSAD9852(SrRedDDSStartNr+2,Frequency);
}
void SetIntensitySrRedMOTSPAOM1(double Intensity) {
  SetIntensityDDSAD9852(SrRedDDSStartNr+2,Intensity);
}
void SetStopFrequencySrRedMOTSPAOM1(double Frequency) {
  SetFrequency2DDSAD9852(SrRedDDSStartNr+2,Frequency);
}
void SetModulationFrequencySrRedMOTSPAOM1(double ModulationFrequency) {
  SetModulationFrequencyDDSAD9852(SrRedDDSStartNr+2,ModulationFrequency);
}
void SetClearACC1SrRedMOTSPAOM1(bool OnOff) {
  SetClearACC1DDSAD9852(SrRedDDSStartNr+2,OnOff);
}
void SetTriangleBitSrRedMOTSPAOM1(bool OnOff) {
  SetTriangleBitDDSAD9852(SrRedDDSStartNr+2,OnOff);
}
void SetFSKBitSrRedMOTSPAOM1(bool OnOff) {
  SetFSKBitDDSAD9852(SrRedDDSStartNr+2,OnOff);
}
```

```
void SetModeSrRedMOTSPAOM1(double Mode) {
  SetModeDDSAD9852(SrRedDDSStartNr+2,Mode);
}
void SetRampRateClockSrRedMOTSPAOM1(double Rate) {
  SetRampRateClockDDSAD9852(SrRedDDSStartNr+2,Rate);
}
```

In the constructor of `IOList.cpp` you have to register:

```
RegisterAnalogOutput(&SetIntensitySrRedMOTSPAOM1,"SetIntensitySrRedMOTSPAOM1",
  "Intensity MOT Red slave 1 SP AOM1 202 (DP)","0..100%");
RegisterAnalogOutput(&SetStartFrequencySrRedMOTSPAOM1,
  "SetStartFrequencySrRedMOTSPAOM1",
  "Start Frequency MOT Red slave 1 SP AOM1 202 (DP)","MHz");
RegisterAnalogOutput(&SetStopFrequencySrRedMOTSPAOM1,
  "SetStopFrequencySrRedMOTSPAOM1",
  "Stop Frequency MOT Red slave 1 SP AOM1 202 (DP)","MHz");
RegisterAnalogOutput(&SetModulationFrequencySrRedMOTSPAOM1,
  "SetModulationFrequencySrRedMOTSPAOM1",
  "Modulation Frequency MOT AOM1 101(DP)","MHz");
RegisterAnalogOutput(&SetModeSrRedMOTSPAOM1,"SetFSKModeSrRedMOTSPAOM1",
  "Set FSK Mode MOT AOM1 101(DP)","0..4");
RegisterAnalogOutput(&SetRampRateClockSrRedMOTSPAOM1,
  "SetRampRateClockSrRedMOTSPAOM1",
  "Set Ramp Rate Clock Sr Red MOT Red slave 1 SP AOM 1","");
RegisterDigitalOutput(&SetClearACC1SrRedMOTSPAOM1,
  "SetClearACC1SrRedMOTSPAOM1","Clear ACC 1 MOT AOM1 101(DP)");
RegisterDigitalOutput(&SetTriangleBitSrRedMOTSPAOM1,
  "SetTriangleBitSrRedMOTSPAOM1","Set Triangle Bit MOT AOM1 101(DP)");
RegisterDigitalOutput(&SetFSKBitSrRedMOTSPAOM1,
  "SetFSKBitSrRedMOTSPAOM1","Set FSK Bit MOT AOM1 101(DP)");
```

And you can start a triangular frequency modulation between the start and stop frequencies in `CSequence::InitializeSystem` or elsewhere during the sequence by:

```
SetClearACC1SrRedMOTSPAOM1(false);
SetTriangleBitSrRedMOTSPAOM1(false);
SetFSKBitSrRedMOTSPAOM1(false);
SetModeSrRedMOTSPAOM1(2);
SetRampRateClockSrRedMOTSPAOM1(1);
SetStartFrequencySrRedMOTSPAOM1(SrRedMOTSPAOM1StartFrequency);
SetStopFrequencySrRedMOTSPAOM1(SrRedMOTSPAOM1StopFrequency);
SetModulationFrequencySrRedMOTSPAOM1(SrRedMOTSPAOMModulationFrequency);
SetIntensitySrRedMOTSPAOM1(0);
WriteMultiIOBus();
SetTriangleBitSrRedMOTSPAOM1(true);
```

After the modulation has started, it is sufficient to modify the start, stop and modulation frequencies to change its properties. It is also possible to perform single, non repeating frequency sweeps. For a detailed description of the DDS programming see the AD9852 datasheet.

4.3.4 Adding a AD9858 DDS

The declaration of DDS frequency command is placed in `IOList.h`.

```
extern void SetFrequencySrBlueMOTSPAOM(double Frequency);
```

The units of the frequency is MHz.

The definition of the frequency command is placed in `IOList.cpp`.

```
void SetFrequencySrBlueMOTSPAOM(double Frequency) {
  Output->RangeCheck(Frequency,/*Min*/60,/*Max*/80,"SetFrequencySrBlueMOTSPAOM");
  SetFrequencyDDSAD9858(/*SoftwareAddress*/0,Frequency);
}
```

The software addresses are given to the AD9858 DDS in the order in which they are registered in the `CSequence::ConfigureHardware()` procedure in chapter 3. The AD9858 software addresses are independent of the AD9852 software addresses.

The DDS output procedures need to be registered in the system. This is done in the constructor of `CIOList` as follows.

```
RegisterAnalogOutput(&SetFrequencySrBlueMOTSPAOM,"SetFrequencySrBlueMOTSPAOM",
  "Sr blue MOT (SP) Frequency","MHz","Some help text",/*Min*/60,/*Max*/80,
  /*Constant*/false,/*Color*/ColorRed);
```

The parameters of the `RegisterAnalogOutput` procedure are described in the 4.1 section.

Each DDS has to be initialized. The initial frequency is set in `CSequence::InitializeSequence`

```
SetFrequencySrBlueMOTSPAOM(SrBlueMOTSPAOMFrequency);
```

The parameters are declared as described in section **??**. A frequency of 0 MHz will stop the DDS and produce a constant DC output. Since the DDS is usually used AC coupled, the signal will drop to 0V. This can be used to switch the radio frequency off.

4.3.5 AD9858 DDS configuration

AD9858 DDS are microcontrollers with many options. You can access every option by directly accessing the DDS using commands provided by the `CAD9858` class. If you are not using the waveform generation mode, you can access these commands like this:

```
AD9858[0]->Set2GHzDividerDisable(On);
WriteMultiIOBus();
```

`AD9858[]` is a table containing pointers to instances of the `CAD9858` class. It is indexed by the software address of the DDS. All possible commands can be found by looking up the `CAD9858` class. The explanation of the commands is given in the AD9858 datasheet, available on the analog devices webpage.

4.4 Servo motors

Servo motors are used for mechanical shutters or for rotable waveplates. The servo motor rotates to a position proportional to the pulse width of a TTL signal send to it. Pulse widths of 0.9 ms and 2.1 ms correspond to minimum and maximum rotation angle respectively. The pulse has to be repeated every 20 ms for the servo to move into place and hold its position with its maximum torque. Once the servo has reached its final position, the position is held by friction even with no pulse signal applied. Our laser beam shutters need 100 ms to change position from open to close. Our rotable waveplates need 150 ms for a 90deg rotation and the oven shutter needs 250 ms from open to closed. Servo motors require a 4.8 V supply that is able to provide sufficient current during servo motion. If the current is sufficient can be determined by measuring the voltage drop of the supply signal during servo motor motion.

The pulse signal is delivered by a digital output. It is of course not sufficient for this output to be set to TTL high for a shutter to move into the open position. Instead a pulse sequence has to be generated. This is done by the `PositionServo` procedure defined in `IOList.cpp`.

The code additions required to add a shutter are given in the following.

In `IOList.h` you need to add two digital outputs per shutter:

```
extern void SwitchSrBlueMOTShutter(bool OnOff);
extern void SwitchSrBlueMOTShutterServoSignal(bool OnOff);
```

The `SwitchSrBlueMOTShutter` digital output is a virtual digital output. It has no corresponding hardware digital output. Setting this digital output to `On` will open the shutter and setting it to `Off` will close the shutter. The `SwitchSrBlueMOTShutterServoSignal` digital output corresponds to the hardware digital output that is connected to the servo motor.

In `IOList.cpp` you need to define these output procedures:

```
const int SwitchSrBlueMOTShutterServoSignalChannelNr=36;
void SwitchSrBlueMOTShutterServoSignal(bool OnOff) {
  Output->DigitalOutScaled(SwitchSrBlueMOTShutterServoSignalChannelNr,OnOff,OnOff);
}
void SwitchSrBlueMOTShutter(bool OnOff) {
  PositionServo(SwitchSrBlueMOTShutterServoSignalChannelNr,OnOff,
    /*OnPulseDuration*/2.0,/*OffPulseDuration*/1.5,
    /*PulseSequenceDuration*/100);
  Output->DigitalOutScaled(HelpDigitalOutStartNumber+
    SwitchSrBlueMOTShutterServoSignalChannelNr,OnOff,!OnOff);
}
```

`SwitchSrBlueMOTShutterServoSignalChannelNr` contains the software address of the digital output used. `SwitchSrBlueMOTShutterServoSignal` is defined as any other digital output. `SwitchSrBlueMOTShutter` is a virtual digital output. Its software address is `HelpDigitalOutStartNumber+SwitchSrBlueMOTShutterServoSignalChannelNr`, which is outside the range of the software addresses of hardware digital outputs since `HelpDigitalOutStartNumber=11000`. The `Output->DigitalOutScaled` does not send a signal to a real output, but it helps to keep track of the status of the virtual digital output and enables control of this output through the manual operation menus.

The output signal is generated by the `PositionServo` procedure. As parameters it requires the specification of the pulse widths that corresponds to the on and off positions of the shutter and the total duration for which the pulse signal should be generated for the shutter to safely move from on to off position. The pulse widths depend on the type of the servo and are usually in the range 0.9 ms to 2.1 ms. The total duration depends on the servo and the load attached to it. It is 100 ms for laser beam shutters, 150 ms for rotable waveplates and 250 ms for the oven shutter. To find the values required for a specific shutter, some servo motor test procedures are accessible through the utility menu. Under the menu item `"Test Servo Digital Output Nr"` you can specify the software address of the digital output to which the shutter that you want to adjust is connected. `"On Pulse Duration"` and `"Off Pulse Duration"` are the pulse width that you want to test to switch the shutter on and off. `"Pulse Sequence Duration"` contains the total duration of the pulse sequence. Using the three buttons `"Test Servo On"`, `"Test Servo Off"` and `"Blink Servo"` you can send pulses with the specified parameters to the servo. In that way you can quickly find the correct parameters for a new servo. Once you have found the parameters you enter them into the "PositionServo" procedure.

The servo motor procedures are registered in the constructor of `CIOList` as follows.

```
RegisterDigitalOutput(&SwitchSrBlueMOTShutter,"SwitchSrBlueMOTShutter",
  "Sr blue MOT Shutter");
RegisterDigitalOutput(&SwitchSrBlueMOTShutterServoSignal,
  "SwitchSrBlueMOTShutterServoSignal","Sr blue MOT Shutter Servo Signal",
  "",/*Constant*/true);
```

It is important to forbid user control of the servo motor signal by setting `/*Constant*/` to `true` in the registration of `SwitchSrBlueMOTShutterServoSignal`.

The shutter needs to be initialized in `CSequence::InitializeSequence()`

```
SwitchSrBlueMOTShutter(On);
```

To easily find broken or badly aligned shutters it is useful to be able to move all shutters at the same time using the `Blink shutters` menu option. To enable this, you have to add the shutter to the `CSequence::SwitchShutters()` procedure. You do this by adding the following line to this procedure:

```
SwitchSrBlueMOTShutter(OnOff);
```

4.5 Duplicating an output signal

It is possible to send a copy of an output signal to another output (analog or digital) for debugging purposes. This works for any type of output and is achieved in the following way:

```
void SetMOTCoilCurrent(double Current) {
    double Voltage=Current*10.0/200.0;
    SwitchOscilloscopeTrigger(Current>100);
    Output->AnalogOutScaled(/*software address*/2,/*unscaled value*/Current,
      /*scaled value*/Voltage);
 }
```

`SwitchOscilloscopeTrigger()` is a digital output declared and registered in the usual way. As soon as the MOT coil current goes above 100 A, this trigger signal is switched on, otherwise it is switched off. It is important that the `SwitchOscilloscopeTrigger()` procedure is called before the `Output->AnalogOutScaled()` is called. If you would change the order an error message would be displayed stating that you tried to use the output used in `SwitchOscilloscopeTrigger()` twice. This method of duplicating an output signal is especially useful to debug DDS frequency and intensity commands since those exist only in digital form and can not be displayed on an oscilloscope. I found it quite useful to check the timing of probe beam light pulses with respect to camera trigger and readout signals in this way (e.g. for fast kinetics imaging).

4.6 Inputs

Usually ultracold atom experiments do not require sophisticated digital or analog inputs. Most often the only analog input required is an input that is connected to a photodiode measuring the fluorescence of the magneto-optical trap (MOT). The experimental sequence is started as soon as the MOT fluorescence has reached the desired value. This task does not require a fast or very precise analog input. For this reason the input side of the control system is very basic.

4.6.1 Digital inputs

The control system supports two types of digital inputs: the inputs of the NI67xx cards or the parallel port (printer port) of the PC. In principle it is possible to use the NI6533 card and the MultiIO system connected to it bidirectionally. But we did not develop any input hardware for it so far. Inputs are accessed directly using

```
bool InputBit=output->DigitalIn(/*Nr*/0,/*BitNr*/0);
```

or

```
unsigned short InputBitPattern=output->DigitalIn(/*Nr*/);
```

`Nr` is the number of the digital input card as defined by the order of input cards in `CSequence::ConfigureHardware()` and `BitNr` is the input bit of this card. The NI67xx card has only 8 bits, with numbers 0..7. The parallel port needs to be configured in bidirectional mode. This can usually be done in the BIOS of the computer. The parallel port has 13 usable bits and this is the pinout:

```
//Parallel port input bits:
//bit 15 = status bit 7 = Pin 11 = Busy (hardware inverted)
//bit 14 = status bit 6 = Pin 10 = Ack
//bit 13 = status bit 5 = Pin 12 = Paper
//bit 12 = status bit 4 = Pin 13 = Select
//bit 11 = status bit 3 = Pin 15 = Err
//bit 10 = not used
//bit  9 = not used
//bit  8 = not used
//bit  7 = data bit 7 = Pin 9
//bit  6 = data bit 6 = Pin 8
//bit  5 = data bit 5 = Pin 7
//bit  4 = data bit 4 = Pin 6
//bit  3 = data bit 3 = Pin 5
//bit  2 = data bit 2 = Pin 4
//bit  1 = data bit 1 = Pin 3
//bit  0 = data bit 0 = Pin 2
```

In the "system configuration" menu the button "Test digital input" calls a procedure that displays the status of the bits of input card 0, which is normally the parallel port.

4.6.2 Analog inputs

Analog inputs are provided by the comparator analog input circuit presented on the control system webpage or by a NI6024E card. In the following we will first describe how to connect a comparator analog input circuit to the system and then explain how to use analog inputs.

Connecting a comparator analog input card

Analog inputs provided by the comparator circuit available for this task on our webpage (www.nintaka.com) work differently. The circuit compares the voltage U_{in} at its input with a reference voltage U_{ref} provided by an analog output, which is connected to the second port of the comparator. If $U_{ref} > U_{in}$ TTL high is sent to a digital input D_{in} of the computer, otherwise TTL low. The computer uses a binary search algorithm to find U_{in}, which can be in the range -10 V to 10 V. To give an example, suppose that $U_{in} = 2.5\,\mathrm{V}$. The computer starts with $U_{ref} = 0\,\mathrm{V}$ and obtains TTL high on D_{in}. This means that U_{in} is somewhere between 0 V and 10 V. This interval is divided in two by comparing U_{ref}=5 V with U_{in} in a second round. In each round the maximum deviation of U_{in} to U_{ref} is divided by two. The maximum number of rounds is limited by the digits of the analog reference output, which usually is 16. Precision can be traded for time by reducing the number of rounds.

Our comparator analog input cards contain a multiplexer, which allows to select one of eight analog inputs. The multiplexer requires three digital signals to specify the analog input.

To install an analog input card, you have to connect an analog output to its reference input, three digital outputs to its port selection inputs, and you have to connect its digital output to a digital input. You can connect any number of analog input cards to the system.

Analog inputs are not deeply integrated into the control system since they are used only in very simple ways, for example to start the experimental sequence as soon as the magneto-optical trap is loaded to a threshold value. The small procedures needed to make these inputs work can be found in `IOList.cpp`. For each analog input card, you have to add an analog output named for example `SetComparatorAnalogInVoltageX`, where `X` needs to be replaced by some string identifying the card (e.g. A,B,C,...). Next you have to add three digital outputs, named `SwitchComparatorAnalogInSourceXY`, where `X` is the same string as before and `Y` is 0,1, and 2 respectively. The analog input is read in the procedure `GetComparatorAnalogIn()` in `IOList.cpp`. It is quite obvious how this procedure works and I will not explain it here any further. You have to place the names you chose for the analog and digital outputs at the obvious places inside the two `switch` statements in this procedure. Finally you have to select the correct digital input for each card in `bool GetComparatorAnalogInResult(int Box)`. It is also possible to use one and the same reference analog output or digital signals for several analog input cards. Only the digital input needs to be distinct for each card.

Adding and using an analog input

To add an analog input based on our comparator card to the system, you place

```
extern double GetSrBlueMOTFluorescence(double AverageTime=0);
```

in `IOList.h` and

```
double GetSrBlueMOTFluorescence(double AverageTime) {
return GetComparatorAnalogInAveraged(/*CardNr*/0,
            /*ChannelNr*/0,AverageTime);
}
```

into `IOList.cpp`. `CardNr` and `ChannelNr` are evidently the number of the analog input card (corresponding to how you configured it in `GetComparatorAnalogIn()`) and the number of the analog input of this card. For a NI6024E card analog input you use instead

```
double GetSrBlueMOTFluorescence(double AverageTime) {
return output->AnalogIn(/*Nr*/0);
}
```

where `Nr` is the software address of the analog input on your system.

You register the analog input with

```
RegisterAnalogInput(&GetSrBlueMOTFluorescence,"Sr blue MOT Fluorescence",
  /*MeasureAfterLoading*/true);
```

at the beginning of `CIOList::CIOList()`, just after

```
for (int i=0;i<NrAnalogInBoxes*8;i++) AnalogInChannelName[i].Format("NC");
```

`MeasureAfterLoading` tells the system to read this input after the fluorescence stopped loading has reached its threshold value, see `bool FluorescenceTrigger(CWnd* parent)` and `CSequence::MeasureAnalogInputValues(bool AfterLoading)` in `Sequence.cpp`. The values of the analog inputs are displayed in the "General parameters" menu. You can read the input with

```
double BlueMOTFluo=GetSrBlueMOTFluorescence();
```

For the time being input commands can not be used while an experimental sequence is running. But if this would be ever required, it would be simple to implement this for digital input ports and NI6024E analog inputs using the technique that is used for synchronizing serial port commands and GPIB commands with the waveform output.

4.7 Serial and GPIB devices

Serial and GPIB devices are connected to the computer via the RS232 port or the GPIB port. A GPIB port is also a type of serial port and requires a national instruments GPIB card. A large number of devices can be connected to a single GPIB port, whereas only one device can be attached to a serial port. To circumvent this limitation for the serial port, we have developed a serial port multiplexer circuit; see www.nintaka.com. Sending commands to or receiving data from serial port or GPIB devices works essentially in the same way. Commands can be either issued before an experimental sequence starts or synchronized to the sequence with a timing precision of about 1 ms.

In the following we explain how to connect the serial port multiplexer, how to connect and use a new serial port or GPIB device and how to write the class describing a serial port or GPIB device.

4.7.1 Serial port multiplexer

Serial port devices can either be connected directly or through the serial port multiplexer (see www.nintaka.com) to the PC. Each multiplexer selects one of eight serial port connectors on its device side to be connected to one serial port of the PC. Serial port multiplexers can be used in series to select between even more devices. Each multiplexer needs three digital signals to select which serial port connector should be connected to the PC at a given time. You need to add three digital outputs to the system, named for example `SetSerialPortSubPortBitX`, where `X` stands for 0,1, and 2. These digital outputs have to be connected to the serial port multiplexer. Next you have to modify the

```
void SetSerialPortSubPort(int Port, unsigned char SubPort)
```

procedure in `IOList.cpp` to configure the necessary digital outputs to access a certain device. `Port` specifies the serial port of the computer and is 1 for COM1, 2 for COM2 etc. `SubPort` is the number of the serial port multiplexers output connector. In the case of a single serial port multiplexer this ranges from 0 to 7. `SetSerialPortSubPort` stores the last port and subport accessed and does not update the digital lines if the same port is accessed again.

4.7.2 Serial port devices

To add a serial port device to the system, you need to create a class derived from `CSerialDevice`, which contains the commands to communicate with the device. How this is done is described in section 4.7.4. Once you have created such a class, you can add devices of this type to the system. In the following we describe the necessary steps for the example `CVerdiLaser`. In `IOList.h` you add

```
class CVerdiLaser;
```

and declare a pointer to an instance of the `CVerdiLaser` class.

```
extern CVerdiLaser* VerdiLaser;
```

In `IOList.cpp` you make sure the header file of your serial port class is included.

```
#include "VerdiLaser.h"
```

You also define the pointer

```
CVerdiLaser* VerdiLaser;
```

and finally in `CIOList::CIOList()`

```
VerdiLaser=new CVerdiLaser(/*COM*/1,/*SubPort*/3,/*speed*/4800,"VerdiLaser");
RegisterSerialDevice(VerdiLaser);
```

`COM` is the number of the serial port of the computer to which the device is connected, labeled starting from 1. If this value is `SerialNotConnected` commands reading from or writing to this device are not executed. `SubPort` is the subport of the serial port multiplexer to which the device is connected. If the serial port multiplexer is not used, this number does not matter. `speed` is the baud rate. `VerdiLaser` is the name of the device displayed in the menu. `RegisterSerialDevice(VerdiLaser);` registers the serial port device.

Commands can be sent to the serial port device using commands similar to

```
VerdiLaser->SetOutputPower(VerdiLaserPower);
```

`SetOutputPower` is one of the methods you created when you wrote the `CVerdiLaser` class; see section 4.7.4. Serial port commands can be executed instantly, or synchronized with the experimental sequence; see 7.6.

4.7.3 GPIB devices

GPIB devices are added in a very similar way to serial port devices. First you derive a class from `CGPIBDevice` containing the commands for your GPIB device. We take as an example the `CKeithley2000` class. You have to assure that

```
class CKeithley2000;
```

is contained at the beginning of `IOList.h`. For each device you declare a pointer that class. In this example we have `NrKeithleyMultimeter` Keithley2000 devices.

```
const unsigned int NrKeithleyMultimeter=2;
extern CKeithley2000* KeithleyMultimeter[NrKeithleyMultimeter];
```

which you define in `IOList.cpp`

```
CKeithley2000* KeithleyMultimeter[NrKeithleyMultimeter];
```

In `IOList.cpp` you also make sure the header file of your GPIB device class is included.

```
#include "Keithley2000.h"
```

Finally you create the instance of `CKeithley2000` in the constructor `CIOList::CIOList`

```
for (int i=0;i<NrKeithleyMultimeter;i++) {
  KeithleyMultimeter[i]=new CKeithley2000(/*GPIBPort*/16+i,
    /*Name*/"Keithley2000"+itos(i),/*MenuText*/"Keithley Multimeter "+itos(i));
  RegisterGPIBDevice(KeithleyMultimeter[i]);
}
```

`GPIBPort` is the port of each device. In this case they have been given GPIB ports 16 and 17. If a device is disconnected, the address can be set to `GPIBNotConnected`. Then all commands issued to the device are ignored. `Name` is a short string giving the device a unique name. `MenuText` is displayed in the menu. `itos` is converting an integer to a string. The `RegisterGPIBDevice` command registers the GPIB device.

Commands like

```
double KeithleyData=KeithleyMultimeter[1]->GetData(KeithleyMultimeterData[1]);
```

communicate with the GPIB device. The commands have been defined by you when writing the class describing the GPIB device, in this example `CKeithley2000`. If you are using a GPIB device for which the describing class already exists, you can find the commands which have been implemented by taking a look at the describing class.

4.7.4 Creating the class describing a serial port or GPIB device

The class describing a serial or GPIB device contains commands to communicate with the device. To create a class for a new device you copy the files containing the class of a similar device (e.g. you copy "Keithley2000.h" and "Keithley2000.h" for a new GPIB device or "VerdiLaser.h" and "VerdiLaser.cpp" for a new serial port device). You rename those files with the name of your new device (e.g. "NewDeviceName.h" and "NewDeviceName.cpp"). You add those two new files to the project by opening the "project explorer" (see the "view" menu). In the project explorer, you click right on "header files", choose "add existing item" and select the "NewDeviceName.h" file. You do the same under "source files" for the "NewDeviceName.cpp" file. Now you open the files and replace just in those files the old device name by your new device name (e.g. search "Keithley2000" everywhere and replace it by "NewDeviceName"). Then you go to the class view. The newly created class "CNewDeviceName" has appeared in the "Control" folder. Right-click on the newly created class, copy it and place the copy for convenience into the "Serial" or "GPIB" folder of the classview. The communication parameters for a serial port device are set in the constructor of its class. You delete the methods corresponding to commands specific to the Keithley2000 or VerdiLaser device. Then you create new commands for your device. Each command is a method of "CNewDeviceName". By taking a look at "CKeithley2000" or "CVerdiLaser" or other such classes, you will find plenty of self-explaining examples of how to do this. Here is one:

```
bool CVerdiLaser::GetOutputPower(double &Power) {
Power=0;
CString buf;
Flush();//empty nput buffer
if (!WriteString("?P;")) return Error("CVerdiLaser::GetOutputPower : error during ?P");
if (!ReadString(buf)) return Error("CVerdiLaser::GetOutputPower : timeout during ?P");
buf=buf.Right(buf.GetLength()-3);
Power=atof(buf);
return true;
}
```

4.8 TCP/IP devices

We usually communicate with two devices over TCP/IP: the data acquisition program and the oven controller. Communication with those devices is only allowed in direct output mode and not integrated into the manual control menus (unless you specify a virtual output to do this). Programming a new TCP/IP device resembles programming a new serial device. You copy `ovencontrol.h` and `ovencontrol.cpp`, rename those files and the `COvenControl` class. Then you add methods communicating with the device. You can see how this is done by looking at the examples in `COvenControl` or `CVision`. Just one simplified example is given here:

```
bool COvenControl::SetTemperature(double Temperature) {
  if (!Connected) return true;
  bool ok=Command("SetTemperature");
  if (ok) {
    WriteDouble(Temperature);
    return true;
  }
  return false;
}
```

You create an instance of this new class for each TCP/IP device of this type that you have and call the `ConnectSocket` of this class for example in the constructor of `CIOList` specifying the IP address and port.

```
OvenControl.ConnectSocket("138.232.183.172",701);
```

You communicate with the device with the methods that you have created.

```
OvenControl.SetTemperature(20/*C*/);
```

4.9 Disabling access to a subsystem

Sometimes it is useful to be able to switch computer control of one laser system off, but be able to run the experiment with the remaining laser systems. In our lithium-potassium mixture experiment, one could in that way run the machine with lithium and change the potassium laser system without being disturbed by the computer changing the AOMs or shutters. To have this option, the following code needs to be inserted into each output command concerning AOMs or shutters of the potassium laser system.

```
if (RunningExperimentalSequence && DisableKLaserControl) return;
```

`DisableKLaserControl` is a bool parameter declared by the user as described in chapter 5. `RunningExperimentalSequence` is `true` is an experimental sequence is executed and `false` in manual mode.

Chapter 5

Parameters

The program provides a system with which the user can declare and register parameters, for example parameters upon which the experimental sequence depends. These parameters are automatically stored on hard disk and are transferred to the data acquisition computer after each experimental sequence. A set of experimental runs can be defined with a few clicks of the mouse by selecting a few parameters to be varied and their variation range. Parameters are global variables. The can be defined for example in "ParamList.cpp", "UtilityDialog.cpp", or "Sequence.cpp". In the following parameters defined in "ParamList.cpp" are used as an example. For parameters defined in "Sequence.cpp" the declaration part can be omitted. See also sections 8.1 and 8.2.

5.1 Adding a parameter

During the registration the user defines the valid parameter range, a description, the units of the parameter, and optionally a help text and the color in which the parameters name is displayed. The following example shows how to add parameters of each of the possible types.

In the file `CParamList.h` you add:

```
extern double MOTCurrent;
extern double LastMOTFluo;
extern bool DoMagneticTrap;
extern long NrCycles;
extern *CString DebugFileName;
extern *CString DetectionMethod;
```

In the file `CParamList.cpp` you add:

```
double MOTCurrent;
 double LastMOTFluo;
bool DoMagneticTrap;
long NrCycles;
*CString DebugFileName;
*CString DetectionMethod;
```

and in the same file, in the constructor of the class `CParamList`, you add:

```
//Arguments:   pointer to variable, Name, Minimum,
//             Maximum, question/description, units, help, color
RegisterDouble(&MOTCurrent,"MOTCurrent",/*Min*/0,/*Max*/50,"MOT current","I",
  "optinal help text",/*optional color*/ColorRed);
RegisterDoubleConstant(&LastFluo,"LastFluo",/*Init*/0,"Last MOT  fluo","V",
  "optinal help text",/*optional color*/ColorRed);
RegisterInt(&NrCycles,"NrCycles",/*Min*/0,/*Max*/10,"Nr Cycles","",
  "optional help",/*optional color*/ColorRed);
//Arguments:   pointer to variable, Name,  length of
//             string, question/description, help, color
RegisterString(DebugFileName,"DebugFileName","C:\\Rubidium\\Debug.dat",
    /*Length*/200,"Debug filename","optional help",/*optional color*/ColorRed);
RegisterStringComboBox(DetectionMethod,"DetectionMethod","Absorption image",200,
  "Detection method",DetectionMethodStringList,/*optional color*/ColorRed);
//Arguments:   pointer to variable, Name, question,
//             short description, optional Radio button ID,
RegisterBool(&DoMagneticTrap,"DoMagneticTrap","Do Magnetic Trap ?",
  /*ShortDescription*/"M",/*optional RadioButtonID*/0,"optional help",
  /*optional color*/ColorRed);
```

Help text and colors are always optional. If you need more colors, add them to `color.h` and `color.cpp`.

5.2 long

The only type of integer parameter allowed is the `long` type. This type is registered using the `RegisterInt` command.

5.3 double

Two types of double variables can be registered. `RegisterDouble` registers a normal double parameter that the user can modify in the menu. `RegisterDoubleConstant` registers a double variable that is displayed in the menu, but can not be modified by the user. This can be used to display values acquired by the program, e.g. the voltage of the MOT fluorescence photodiode at the beginning of the last experimental sequence.

5.4 bool

Three different kinds of bool variables exist. If a short description is given during the registration of a `bool` variable, the check box of this variable appears left in the menu, otherwise right. My standard is to use check boxes displayed left for bool variables that switch whole code blocks of the experimental sequence on or off. Variables with check boxes displayed right only slightly modify the behavior of a code block. This will be described in more detail in section ??. A third type of bool variables are bool variables grouped together in a radio-button group box. If one of the buttons inside the group is selected, the others are automatically deselected. The group box is a box drawn around the group of buttons with a title displayed on top. Each menu panel can only contain one such group. Here is an example that shows the creation of two radio buttons inside a group box.

```
StartGroupBox("Imaging Configuration");
```

```
RegisterBool(&UseNormalImaging,"UseNormalImaging","Use normal imaging","",IDB_RADIO);
RegisterBool(&UseFutureOptionImaging,"UseFutureOptionImaging",
  "Use other imaging (future option)","",IDB_RADIO);
StopGroupBox();
```

5.5 CString

Two kinds of string variables exist: normal strings typed into an edit box by the user (see `RegisterString` example) and strings selected from a predefined list of strings (see `RegisterStringComboBox` example). The string list for the `RegisterStringComboBox` command is created as follows. In the class definition in `IOList.h` or `ParamList.h` you define

```
CStringList *DetectionMethodStringList;
```

Inside the constructor `CIOList::CIOList` or `CParamList::CParamList` before the `RegisterStringComboBox` command you create the list.

```
DetectionMethodStringList=new CStringList();
DetectionMethodStringList->AddHead("Fluorescence image");
DetectionMethodStringList->AddHead("Absorption image");
DetectionMethodStringList->AddHead("Fast kinetics absorption image");
```

In the destructor `CIOList::~CIOList()` or `CParamList::~CParamList()` you delete the list

```
delete DetectionMethodStringList;
```

Chapter 6

Menus

The layout and behavior of menus can be influenced with additional commands inserted in between registration commands in the constructor `CIOList::CIOList`, `CParamList::CParamList`, `CUtilityDialog::CUtilityDialog`, or `CSystemParamList::CSystemParamList`. In addition it is possible to create buttons in the menus that call methods of `CSequence`. This can for example be used to call small experimental sequences independent of the main experimental sequence.

6.1 NewMenu

A new menu panel is started with the command

```
NewMenu(/*Title*/"Optical dipole trap parameters",/*MessageID*/0,/*ModeOfMenu*/0,
  /*Color*/ColorRed);
```

Only the `title` parameter is required, all others are optional. `MessageID` is a message ID that is sent to `CSequence::MessageMap` each time the user leaves this menu. If `MessageID` is zero, no message is sent. See `AddButton` for more information. `ModeOfMenu` defines the storage mode with which the subsequent parameters are handled. If `ModeOfMenu=0` the parameters are displayed in the menu, stored on hard disk, and send to the data acquisition program after each run of the experiment. If `ModeOfMenu=1` the parameters are displayed in the menu and stored on hard disk. If `ModeOfMenu=2` the parameters are only stored on hard disk.

6.2 NewColumn

The `NewColumn` command

```
 NewColumn();
```

starts a new column in the display of parameters in the menu panel.

6.3 Static text

A line of text can be generated by

```
AddStatic("life is beautiful","optional help text",/*Color*/ColorGreen);
```

This can also be used to create an empty line:

```
AddStatic("");
```

6.4 Linking experimental sequences to buttons on the user interface

All experimental sequences are defined in methods of the CSequence class. For the main experimental sequence, the DoExperimentalSequence method is already prewired to the ”Run Experimental Sequence” button. But perhaps you would like to add some helper or test sequences. For this you need to create a button on the user interface and link it to your sequence. To put a button on the user interface, you go to the Visual C++ resource editor and add a String resource, for example:

```
ID: IDM_TEST_SEQUENCE
Caption: Test sequence
```

After this, IDM_TEST_SEQUENCE is a unique number, identifying the link between the button and the sequence.

Now you add the button to a menu. If the button should appear in the manual control menus, then you add

```
AddButton(IDM_TEST_SEQUENCE,&Sequence);
```

into the constructor of the class that describes the menus you want to have the buttons added to (e.g. `CIOList::CIOList` for buttons in the manual control menus, `CParamList::CParamList` for buttons in the list of parameters and `CUtilityDialog::CUtilityDialog` for buttons in the utility menus). In case you would like to have the button in the main menu, you have to add

```
AddElement(new CElementButton(IDM_TEST_SEQUENCE,&Sequence));
```

into `CMainDialog::CMainDialog`.

`&Sequence` is the destination of a message with the message ID `IDM_TEST_SEQUENCE` that the button sends out to the rest of the program. The `MessageMap` method of `Sequence` will receive the message. You create a new method of `CSequence` by rightclicking with the mouse on `CSequence` in the workspace window and selecting ”add member function”. Choose a function of type ”void” and call it `TestSequence`. `Sequence.h` and `Sequence.cpp` are automatically modified as necessary and the new function appears at the end of the `sequence.h` file:

```
void CSequence::TestSequence() { }
```

Inside this function you add your new sequence, eg:

```
void CSequence::TestSequence() {
   SwitchOvenShutter(Off);
}
```

Now you have to link this sequence to the message map by adding this line to it:

```
bool CSequence::MessageMap(unsigned int Message,CWnd* parent) {
   bool Received=true;
   //stuff here
   switch (Message) {
       //stuff here
       //your new line:
       case IDM_TEST_SEQUENCE: TestSequence(); break;
       default: Received=false;
   }
   //stuff here
   return Received;
}
```

6.5 Submenus

You can create buttons opening submenus using

```
AddDialogButton(CString aText,CEasyDialog* aSpawnDialog,const CString &Help="",
  const COLORREF Color=RGB(1,1,1));
```

The creation of the `CEasyDialog` class `aSpawnDialog` requires a bit more advanced programming. You can take a look at classes derived from `CEasyDialog` to learn how to do this, especially classes derived from `CParamDialog`, like `CUtilityMenu`. Searching the code for `AddDialogButton` is another way of finding examples.

Chapter 7

Programming sequences of input/output commands

This chapter explains how to implement sequences of commands. This style of programming is used for simple code sequences and for the main experimental sequence. Some additional structures concerning only the main experimental sequence are the topic of chapter 8.

7.1 Modes of command execution

Commands can be executed in two different modes, called direct mode and waveform mode.

The **direct mode** is the default mode. If the program encounters a write command like `SetMOTCurrent(0);` this command is instantly executed. The advantage of the direct mode is its simplicity and its ability to mix input and output commands. This ability makes the direct mode useful to prepare the system and start an experimental sequence as soon as for example the fluorescence of the magneto-optical trap has reached a desired value. The main drawback of this mode is the poor timing precision on the order of 1 ms.

The **waveform mode** has a timing precision which is as good as the clock used for the output cards, typically better than 100 ns. To write a sequence of commands, a table with output values for each clock cycle and each output card is generated. After enough of this table has been calculated the output process is started. For long sequences of commands, the calculation of the table of output commands is continued while the output process is running. Only input commands using input ports on a national instruments card can be synchronized with the output sequence. Even then it is not possible to influence the output sequence by the values just measured.

In addition to the direct mode and the waveform mode, several other modes exist that do not execute commands.

In the **preparation mode** serial port and GPIB devices can be prepared for usage. For example the pulse sequence of a high speed pulse generator could be calculated and transmitted to the device. All other output commands and all timing commands are ignored.

The **assemble sequence list mode** stores commands in a table, called sequence list. This allows time reordering of the output commands using for example the `GoBackInTime` command.

The **memory readout mode** reads the input date acquired during the waveform generation mode.

Some special modes exist that the user should not have to use.

In the **force output mode** every output command is executed, even if the output is already in the state that is requested. Normally commands which seem unnecessary are discarded. This mode is useful to initialize the system a first time when the program is started.

The **scaled out store mode** is used only once, when the program is started. Every output procedure is called once in this mode and the program acquires the software address associated with each output command. All other modes are **scaled out normal modes**.

To enter a certain mode, the following commands can be called in methods of `CSequence`.

```
SetDirectOutputMode();
SetPreparationMode();
SetAssembleSequenceListMode();
SetWaveformGenerationMode();
SetMemoryReadoutMode();
```

Outside of methods of `CSequence Output->` has to be placed in front of each of those commands. Sometimes it is useful to know in which mode the program is. This can be done using the functions

```
Output->IsInDirectOutputMode()
Output->IsInAssembleSequenceListMode()
Output->IsInWaveformMode()
Output->InScaledOutStoreMode()
Output->InOutScaledNormalMode()
```

7.2 Programming simple sequences

Let us start to program a simple sequence of output commands. The following example sequences could be placed in a method of `CSequence` that is called by pressing a button on the user interface; see Sec. 6.4.

Very simple sequences of commands can be executed in direct mode. If the outputs commands should depend on the result of input commands the direct mode is even required. Here as example a simple output sequence that discards atoms from the magneto-optical trap (MOT) until the MOT fluorescence is below a given threshold.

```
while (GetMOTFluorescence()>MaxMOTFluorescence) {
    SwitchMOTLight(Off);
    Wait(1);
    SwitchMOTLight(On);
}
```

Normally it is beneficial to execute the sequence of commands fast and with high timing precision. For this the waveform mode is used. Here an example:

```
SetAssembleSequenceListMode();
StartSequence();
SwitchMOTLight(On);
SetMOTCurrent(MOTCurrent);
Wait(MOTLoadingTime);
StopSequence();
SetWaveformGenerationMode();
ExecuteSequenceList(/*ShowRunProgressDialog*/false);
EmptyNIcardFIFO();
```

The `SetAssembleSequenceListMode()` command tells the system to place each subsequent command into the "sequence list" instead of executing it directly. The commands in this list are executed later in the waveform generation mode. `StartSequence()` is the first command stored in that list. It will start the calculation of the waveform sent to the output cards. `StopSequence()` ends the waveform calculation. `SetWaveformGenerationMode()` switches the waveform generation mode on. `ExecuteSequenceList()` goes through the list of commands and executes each. The last command contained in that list, `StoSequence`, will put the system again into the direct output mode. The optional parameter `ShowRunProgressDialog` determines if the progress dialog displaying the status bars of the waveform generation should be shown. The `EmptyNIcardFIFO()` empties the NI6533 card FIFO. It is not required if the NI6533 card is working correctly. I just once had a damaged NI6533 card that required this command as a work around and since it does not disturb working cards I continued to use it.

7.3 Timing commands

Timing commands are used to separate output commands in time and to reorder the timing sequence of output commands. Time is always specified in milliseconds. The basic timing command is the `Wait(WaitTime)` command. For the amount of time specified the system will wait. This does not necessarily mean that nothing is written on the output ports during that time. Outputs commands might have slipped into that wait period by time reordering commands. And software waveforms might be executed on some outputs; see Sec. 7.5. For debugging purposes, a second, optional parameter called "Wait ID" can be specified

```
Wait(/*Time in ms*/WaitTime,/*optional Wait ID*/1234);
```

The Wait ID simplifies finding the source code responsible for a certain output command; see chapter 10.
The command

```
WaitTillBusBufferEmpty(/*optional Wait ID*/1234);
```

waits till all commands (excluding software waveforms) have been written out. It is useful to place this command before timing critical commands to make sure those timing critical commands are not unnecessarily delayed by previous commands. The sequence

```
SwitchCameraTrigger(Off);
WaitTillBusBufferEmpty(/*optional Wait ID*/1234);
SwitchCameraTrigger(On);
WaitTillBusBufferEmpty(/*optional Wait ID*/1235);
SwitchCameraTrigger(Off);
WaitTillBusBufferEmpty(/*optional Wait ID*/1236);
```

produces the shortest possible pulse. By contrast

```
SwitchCameraTrigger(Off);
SwitchCameraTrigger(On);
SwitchCameraTrigger(Off);
```

would not produce a pulse at all. The `SwitchCameraTrigger(On)` is never executed since it is overwritten by the following `SwitchCameraTrigger(Off)` command.
The command

```
SyncToLine(/*Phase*/0);
```

waits between 0 ms to 1/LineFrequency (20 ms for Europe) to bring the next command into the same phase relation with line as the hardware trigger at the start of the experimental sequence. A delay of `Phase/(360*LineFrequency)` is added. The `SyncToLine` command works only in waveform generation mode. And only if the waveform generation is hardware triggered by a signal in phase with line and the clock signal is also phase stable with line. You can find an electronic circuit to provide those signals on our webpage www.nintaka.com. You also have to set the nominal line frequency of your country in `CSequence::ConfigureHardware` using the `SetLineFrequency` procedure.

7.3.1 Time reordering

Sometimes it is useful to perform an output command before the position where the command appears in the source code. As we will see in Sec. 8.1, the source code is usually segmented in blocks of code dedicated to perform a step in the experimental sequence. Such a step could be the flash of a laser beam. Laser beams are usually blocked by mechanical shutters that take several milliseconds to open. Thus the command to open the shutter needs to be given before the laser beam is used, which usually means, before the code block is reached. To execute the command at the correct time, time reordering commands are used. Here an implementation of the example discussed.

```
void CSequence::LaserFlash() {
    double Time=GetTime();
    GoBackInTime(ShutterPreTriggerTime);
    SwitchFlashShutter(On);
    GoToTime(Time);
    SwitchFlashAOM(On);
    Wait(FlashTime);
    SwitchFlashAOM(Off);
}
```

The time within the sequence (counting from the `StartSequence` command) is retrieved with the `GetTime` function and stored in the local variable `Time`. The `GoBackInTime` command ensures that the following `SwitchFlashShutter` command is placed into the sequence list at the time `Time-ShutterPreTriggerTime`. The `GoToTime(Time)` command ensures that the next commands will be placed into the sequence list as if the `GoBackInTime` command had not happened.

Other time reordering commands exist. `TimeJump(-Delay)` is equivalent to `GoBackInTime(Delay)`. `Delay` can have positive or negative values. `FinishLastGoBackInTime()` can replace the `GoToTime(Time)` command used to return to the current time. It also can undo the last `TimeJump` command. Using `FinishLastGoBackInTime` would make the `GetTime` command and the local variable `Time` unnecessary. But it requires attention from the reader of a piece of source code to find the last `GoBackInTime` command. All those time reordering commands can have an optional `WaitID` parameter.

Sometimes it is useful to perform an action after a certain code block has finished. For example it is often useful to keep AOMs on as much as possible to reduce drifts from temperature changes in AOMs. Here an example that switches the AOM on after the shutter has closed.

```
void CSequence::LaserFlash() {
    double Time=GetTime();
    GoBackInTime(ShutterPreTriggerTime);
      SwitchFlashAOM(Off);
      SwitchFlashShutter(On);
    GoToTime(Time);
    SwitchFlashAOM(On);
    Wait(FlashTime);
    SwitchFlashAOM(Off);
    SwitchFlashShutter(Off);
    Time=GetTime();
      Wait(ShutterClosingTime);
      SwitchFlashAOM(On);
    GoToTime(Time);
}
```

Sometimes two such `LaserFlash` commands have to be executed with a time delay from each other shorter than the time required to move the shutter. In those cases it is best to give the user the full control of what happens with the shutter during the procedure. This is done by introducing additional parameters specifying if the shutter should be opened at the beginning or closed at the end of the code block.

```
void CSequence::LaserFlash(int Nr) {
    double Time=GetTime();
    if (LaserFlashOpenShutter[Nr]) {
        GoBackInTime(ShutterPreTriggerTime);
          SwitchFlashAOM(Off);
          SwitchFlashShutter(On);
        GoToTime(LaserFlashTime);
    }
    SwitchFlashAOM(On);
    Wait(LaserFlashTime[Nr]);
    SwitchFlashAOM(Off);
    if (LaserFlashCloseShutter[Nr]) {
        SwitchFlashShutter(Off);
        Time=GetTime();
          Wait(ShutterClosingTime);
          SwitchFlashAOM(On);
        GoToTime(Time);
    }
}
```

The parameters of this code block now depend on the `Nr` parameter of the procedure. The user would select to open the shutter, but not close it the first time this code block is called. The second time the shutter would be not opened but only closed. More information on how to properly program code blocks will be given in Sec. 8.1.

7.3.2 Timing accuracy

The exact moment in time at which a command is executed can not be known precisely because of several reasons. The bus system has a maximum throughput of 16 bit of data every clock cycle, which is typically every 0.5 μs. Some commands, like DDS frequency commands, contain more than 16 bit of data and are distributed on several clock cycles. The following shows an example where this behavior can lead to a problem.

```
SetMOTLaserDetuning(MOTDetuning);
SetMOTLaserIntensity(MOTIntensity);
```

```
SwitchMOTCoil(On);
SwitchFlashAOM(On);
Wait(0.01);
SwitchFlashAOM(Off);
```

The laser flash should be 10 μs long, but in reality it will be shorter. All output commands before the `Wait` command are here requested to be executed at the same time, which is not possible. The commands are translated into bit patterns sent over the bus and stored on a stack of such commands. It takes some time to transmit this stack. Since the `SwitchFlashAOM` is the last command on the stack, it will be delayed by 7 clock cycles corresponding to 3.5 μs (if a write precision of 3 bytes was specified for the MOT laser DDS). The duration of the flash will be reduced by that time.

To prevent this delay, it is good practice to frame timing critical commands by `WaitTillBusBufferEmpty()` commands:

```
SetMOTLaserDetuning(MOTDetuning);
SetMOTLaserIntensity(MOTIntensity);
SwitchMOTCoil(On);
WaitTillBusBufferEmpty();
SwitchFlashAOM(On);
Wait(0.01);
SwitchFlashAOM(Off);
WaitTillBusBufferEmpty();
```

Here at least the commands directly before and after the `SwitchFlashAOM` will not disturb the duration of the laser flash. But it is still possible that the duration of the flash is changed by additional bus traffic. This traffic could come from time reordered commands from other code blocks, as the shutter commands in the example above. And it can come from software waveforms running in the background. It is the responsibility of the programmer to avoid these situations. If the timing is very critical it is best to check it using an oscilloscope. If you need to know the delay of the additional wait commands, you replace the `WaitTillBusBufferEmpty()` commands by `Wait(0.01)` or similar commands.

7.4 Storing and retrieving output values

It is possible to store the values of output channels and retrieve those values later. This can be useful in some special situations. For example fluorescence measurements. Here, a pair of measurements, one with, one without atoms is taken and the signal from the atoms is extracted. To recreate the background light situation for the background measurement, the setting of the MOT laser beam during the first measurement can be stored. The laser is then briefly switched off to discard the atoms and brought back to the initial value for the background measurement. By storing and retrieving the output value, the measurement procedure will work whatever setting the MOT laser had before the measurement. Another example can be found in Sec. 7.9.

The commands used to store and retrieve the output value are

```
IOList.StoreAnalogOutValue("SetIntensitySrBlueMOTSPAOM");
IOList.RecallAnalogOutValue("SetIntensitySrBlueMOTSPAOM");
```

The `StoreAnalogOutValue` stores the outputs value at that moment in a variable. The `RecallAnalogOutValue` sets the output back to that value. This could for example be used to prepare output

In direct output mode, the value of an analog (or digital) output can be accessed by

```
IOList.GetAnalogOutValue("SetMOTLightIntensity",MOTLightIntensity);
IOList.GetDigitalOutValue("SwitchMOTAOM",MOTAOMStatus);
```

where "SetMOTLightIntensity" ("SwitchMOTAOM") is the analog (digital) output function name and MOTLightIntensity (MOTAOMStatus) is a double (bool) variable to which the value is stored.

7.5 Software waveforms

Often it is required to change an output in a smooth way. This is done using software waveforms. The most common are linear ramps and sinusoidal waveforms. It is easy to implement more waveform types. Software waveforms can only be used in the waveform generation mode.

Here is a simple example for the usage of linear ramps implemented with software waveforms.

```
double TimeStep=0.1;
Waveform(new CRamp("SetFeshbachPSCCurrent",LastValue,RampZSOffFeshbachCurrent,
  ZSRampTime,TimeStep,/*optional: force execution*/true));
Waveform(new CRamp("SetIntensitySrBlueZSSPAOM",LastValue,0,ZSRampTime,TimeStep));
Wait(ZSRampTime);
```

The `Waveform` command puts a waveform in the list of waveforms that are currently executed. As a parameter it takes an instance of a class representing the waveform, in this case `CRamp`, a linear ramp. The constructor of `CRamp` takes as parameters the name of the output on which the waveform is executed, the start value and end value of the output, the duration of the ramp and the time resolution of the ramp. The start value can be replaced by the constant `LastValue`. In that case the waveform starts with the value that the output had just before the waveform did begin. In case this last value is the same as the end value of this ramp the ramp is not executed[1]. Waveforms are never completely smooth since output rate and dynamic range of the outputs are finite. Instead of a linear ramp a staircase pattern is produced. The time resolution of this staircase is specified by `TimeStep`. It should be chosen as large as possible to reduce the computational load. The waveforms are executed during the next `ZSRampTime` milliseconds, in this case during the `Wait` command.

At the end of this wait time, a last value of the waveform might still linger on the stack of the bus system. This can have unwanted consequences. To avoid this problem, use the following programming style:

```
double TimeStep=0.1;
StartNewWaveformGroup();
Waveform(new CRamp("SetFeshbachPSCCurrent",LastValue,RampZSOffFeshbachCurrent,
  ZSRampTime,TimeStep));
Waveform(new CRamp("SetIntensitySrBlueZSSPAOM",LastValue,0,ZSRampTime,
  TimeStep));
WaitTillEndOfWaveformGroup(GetCurrentWaveformGroupNumber());
```

The `StartNewWaveformGroup` function defines that a new waveform group starts. It delivers the number of this waveform group as return value, in case you are interested in that. The `GetCurrentWaveformGroupNumber()` delivers the same number. The `WaitTillEndOfWaveformGroup` command waits till the waveforms belonging to the specified group have finished, including commands from those waveforms that might

[1]This can lead to problems. For example if you want to keep the modulation frequency of a DDS constant while changing the start and stop frequencies of the frequency sweep. If you do not reprogram the modulation frequency, this frequency will change if the boundaries of the frequency sweep interval are changed. To avoid this, you need to reprogram the modulation frequency to always the same value. You can force the waveform to be executed with identical start and end value by setting the optional `force execution` variable to `true`.

have lingered on the bus system stack longer than the duration of the waveform. The uncertainty in the duration of this wait command is the `TimeStep` duration plus about $1\,\mu$ per waveform (in dependence of the nature of the outputs used).

You can also mix these programming styles:

```
double TimeStep=0.1;
StartNewWaveformGroup();
Waveform(new CRamp("SetFeshbachPSCCurrent",LastValue,RampZSOffFeshbachCurrent,
  ZSCurrentRampTime,TimeStep));
Waveform(new CRamp("SetIntensitySrBlueZSSPAOM",LastValue,0,ZSLightRampTime,
  TimeStep));
Wait(ZSLightRampTime);
SwitchZSLightShutter(Off);
WaitTillEndOfWaveformGroup(GetCurrentWaveformGroupNumber());
```

Sometimes it is useful to execute only a fraction of the ramp. For example the evaporative cooling ramp could be interrupted after a certain fraction of it has been executed.

```
double TimeStep=0.1;
StartNewWaveformGroup();
Waveform(new CRamp("SetDipoleTrapAttenuation",LastValue,EndTrapPower,RampTime,
  TimeStep));
if (RampFraction<100) {
    Wait(RampTime*RampFraction/100);
    RemoveWaveformGroup(GetCurrentWaveformGroupNumber());
} else WaitTillEndOfWaveformGroup(GetCurrentWaveformGroupNumber());
```

`RemoveWaveformGroup` removes all waveforms belonging to the group specified from the list of waveforms. Using this programming style, the waveform can be stopped after a certain fraction by just varying one parameter, the ramp fraction, instead of varying the end value and the duration together.

Another usage of the `RemoveWaveformGroup` could be to stop a waveform group in a certain code block, that was started in a different code block and running since then.

The waveforms provided are CRamp, CSineRamp, CParabolicRamp, CSin, CPulse, CRectangle, CSquare, CDelayedWaveform, and CTimeStretch and a short overview of those waveforms is given in the following.

```
CRamp(CString OutputName, double Start, double Stop,
    double Time, double DeltaTime=0, bool ForceExecution=false);

 Meaning of parameters:
  OutputName: Name of output
  Start: start level. if (Start==LastValue) the last value is taken
  Stop: end level
  Time: Duration of ramp
  DeltaTime: how often the output is updated. if zero,
              it is updated as often as possible
  ForceExecution: if true, ramp executed even if Start==Stop

CSineRamp(CString aOutputName, double aStart, double aStop, double
aTime, double aDeltaTime)
  as CRamp, but sine shape with offset and slope.
```

```
CParabolicRamp(CString aOutputName, double aStart, double aStop,
double aTime, double aDeltaTime)
  As CRamp, but constant acceleration and deceleration-> Parabolic
  shape.

CSin(CString OutputName, double Amplitude, double Frequency,
    double Time=0, double Phase=0, double DeltaTime=0);

Meaning of parameters:
  OutputName: Name of output
  Amplitude: Amplitude
  Frequency: Frequency in Hz
  Time: duration, if 0 it is infinite
  Phase: phase. If phase=LastValue, phase is adapted to
        start waveform steadily from old value
  DeltaTime: how often the output is updated.
        if zero, it is updated as often as possible

CPulse(CString OutputName, double LowDelay, double HighDelay,
    long NrPulses=1, int InitLevel=0, bool StayAtEnd=false,
    double AmplitudeLow=0, double AmplitudeHigh=5);

 Meaning of Parameters:
  OutputName: Name of output
  LowDelay: how long to stay low after start
  HighDelay: how long to stay high after transition
  NrPulses: number of pulses, 0 means infinite
  InitLevel: initial level 0: low 1: high
               2: the level the output had before this command
  StayAtEnd: if true, only an edge is produced, not a real puls
  AmplitudeLow: for analog outputs: value corresponding to low
  AmplitudeHigh: for analog outputs: value corresponding to high

CRectangle::CRectangle(CString aOutputName, double aFrequency,
double aDutyCycle,double aAmplitudeLow, double
aAmplitudeHigh,double aTime, bool aStayAtEnd)
 special case of CPuls.

CSquare(CString OutputName, double Frequency, double Time=0,
    int aInitLevel=0,bool StayAtEnd=false, double AmplitudeLow=0,
    double AmplitudeHigh=5);

 Meaning of parameters:
  OutputName: Name of output
  Frequency: Frequency in Hz
  Time: duration, if 0 it is infinite
  InitLevel: initial level 0: low 1: high
        2: the level the output had before this command
  AmplitudeLow: for analog outputs: value corresponding to low
  AmplitudeHigh: for analog outputs: value corresponding to high
```

```
CDelayedWaveform(CWaveform* aMyWaveform, double aDelay)
  Meaning of parameters:
  aMyWaveform: the waveform to be executed
  aDelay: Delay before starting the waveform in milliseconds.

CTimeStretch(CWaveform* aMyWaveform, double aSpeedup, double
aTime)
  Meaning of parameters:
  aMyWaveform: the waveform to be executed
  aSpeedup: initial increase in speed, compensated for by
            subsequent slowdown. Nice for adiabatic ramps, when
            adiabaticity condition changes with time.
  aTime: total time of waveform
```

The `CDelayedWaveform` and `CTimeStretch` examples show waveforms that take other waveforms as parameters and influence their behavior.

It is easy to add more waveforms by creating new waveform classes. To do this you could start by duplicating and renaming `ramp.h` and `ramp.cpp`. Next rename the `CRamp` class in the newly created files and add those file to the project as described in Sec. 4.7.4. The waveform is initialized in the `Init` method, which is only called once when the waveform is started. The output is updated in the `SetOutputs` function. If this function returns `false` the waveform will be deleted from the list of waveforms.

Waveforms can be made dependent on other waveforms. For example in a scanning AOM setup which should stir a laser beam in a circle, the two scanning AOMs should change their frequencies in phase. This can be achieved by defining a virtual analog output, that specifies the phase angle and is ramped linearly upwards. The analog out procedures controlling the AOMs frequencies calculate are made to calculate their frequencies in dependence of this phase angle. Variations of this scheme would allow changes in the ellipticity of the produced trap, rotation of the axis of the average potential and so forth.

Another scheme to complicated waveforms synchronized on several outputs consists of calculating a table of output values during the preparation stage of the experimental sequence. During sequence execution the values of this table are sent out. A software waveform that takes its values from a table needs still to be programmed, which is an easy task.

The calculation of waveforms can be sped up by running the control program in ”Release” mode and not ”Debug” mode. This mode is selected in Visual Studio using the pull down selector in the icon bar next to the green, triangular ”play” button that starts the program.

7.6 Serial port device and GPIB device programming

The configuration of serial port devices and GPIB devices and he format of the commands to those devices was discussed in Sec. 4.7.4. Here we discuss some additional aspects concerning the different programming modes and better integration into the system. In the following ”serial device” will be used as short name for both types of devices.

Commands to serial devices can be used in direct output mode and in waveform generation mode. In waveform generation mode only output commands are allowed since the system can not wait for feedback. In direct output mode all types of commands are allowed. Synchronization of serial device commands with the sequence sent out through the national instruments cards is done by checking the progress of the waveform output from time to time. If the time comes close to the moment a serial device command needs to be sent, the software will stop doing anything else than checking the output progress. As soon as the output time has been reached the serial port command is sent. This usually works with a timing precision of better than 1 ms. The computer will check the progress of waveform output again after the serial command has been sent. If the time laps is more than specified using the `Output->SetMaxSyncCommandDelay(0.01);` command in `CSequence::ConfigureHardware()` (here 10 ms) an error message will be displayed.

Sometimes it is useful to send preparation commands to serial devices while the sequence list is assembled and store only a few commands in the sequence list. An example is the preparation of an arbitrary waveform generator. You would program the waveform beforehand and at the most send a trigger signal over the serial port (in case the waveform generator has no TTL trigger input). The programming looks nicer if the preparation of the generator and the trigger signal can be programmed close to each other in the experimental sequence (e.g. in the same code block, see Sec. 8.1). This also guarantees that the generator is only reprogrammed if needed. To do this, use the following code.

```
//we are in the assemble sequence list mode here.
Serial.SetDirectOutputMode();
//or GPIB.SetDirectOutputMode();
//here commands to prepare the generator before sequence list execution.
Serial.SetStoreInSequenceListMode();
//or GPIB.SetStoreInSequenceListMode();
//here commands to trigger the generator during sequence list execution.
```

Sometimes it is useful to integrate one or a few function(s) of a serial device into the system as if it was a normal analog or digital output. This function can be for example be the frequency or intensity of a frequency generator or the pump diode current of a fiber laser. You would like to be able to change this frequency or intensity using the manual control panels and perform software waveforms on that function (e.g. to perform an evaporative cooling sweep by reducing the power of the dipole trap). After having integrated the serial device as discussed in Sec. 4.7.4, this additional behavior can be achieved by declaring, defining and registering an additional analog output, just as if it was an analog output on a national instruments card or the MultiIO bus system. The only difference occurs is the definition of the output procedure in `IOList.cpp`. It needs to look similar to the following code.

```
void Set100WIRLaserCurrent(double Current) {
  if (Output->InMenuUpdateMode()) return;
  if (!Output->InScaledOutStoreMode()) {
    if (Current>8000) Current=8000;
    IPGLaser[0]->SetOutputCurrent((unsigned int)Current); //in mA
  }
  Output->AnalogOutScaled(HelpAnalogOutStartNumber+2,Current,Current);
}
```

The output can be modified using this procedure. Either directly like `Set100WIRLaserCurrent(1)` or using a software waveform like

```
Waveform(new CRamp("Set100WIRLaserCurrent",LastValue,
  RampIRLaserCurrentCurrent,RampIRLaserCurrentTime,50));
```

Note that the timestep of this waveform has been chosen quite large since the laser needs this time to process the serial port commands. This slowness is typical for serial devices. The

```
if (Output->InMenuUpdateMode()) return;
```

statement is optional. If it is there, the user can not modify this analog output in the manual control menu. Since the 100 W laser is a bit dangerous, this statement is here for security. (In our case this lasers power can be changed in a special menu that also displays information on the laser.) The `if (!Output->InScaledOutStoreMode())` statement hinders the output code from being executed during the startup of the program, when the system is just acquiring the software output channel of this procedure. The last command

```
Output->AnalogOutScaled(HelpAnalogOutStartNumber+2,Current,Current);
```

allows the system to treat the output as if it was a usual analog output on the system. `HelpAnalogOutStartNumber` is the base software address for all such help analog outputs. `HelpAnalogOutStartNumber+2` is the software address and you have to assure that it is unique. To help you with that an error message will be displayed when the program is started and this is not the case. If you would like to implement a digital output in this way, use

```
Output->DigitalOutScaled(HelpDigitalOutStartNumber+2,OnOff,OnOff);
```

instead.

If a serial port multiplexer is used, these new analog or digital output procedures have to be registered in a different manual control menu panel than the digital lines controlling the serial port multiplexer. These lines are adjusted such that signals are sent to the device you are interested in automatically in direct output mode. In waveform output mode you have to take care yourself that these digital lines are set in time to the correct serial port device. If you need to send commands only to one serial port device during the execution of the sequence list you can set the digital lines to the correct device by sending an arbitrary command to that device before the execution of the sequence list and then not sending any command to any other serial port device till the execution is over. If you want to send commands to several serial port devices, you have to place additional commands into the sequence list that put the digital lines into the correct status in time before the serial port command is send. GPIB device commands do of course not have these complications.

7.7 Code example "Position Servo"

With everything we have discussed in this chapter so far, we are able to understand the code used to command servo motors. This is another code examples of a virtual digital output that has no corresponding hardware digital output. And it demonstrates switching to waveform output mode if needed.

In section 4.4 we saw that servo motors are implemented in the following way.

```
const int SwitchOvenShutterServoSignalChannelNr=11;
void SwitchOvenShutterServoSignal(bool OnOff) {
  Output->DigitalOutScaled(SwitchOvenShutterServoSignalChannelNr,OnOff,OnOff);
}

void SwitchOvenShutter(bool OnOff) {
  PositionServo(SwitchOvenShutterServoSignalChannelNr,OnOff,/*OnPulseDuration*/1.5,
    /*OffPulseDuration*/1.9,250);
  Output->DigitalOutScaled(HelpDigitalOutStartNumber+
    SwitchOvenShutterServoSignalChannelNr,OnOff,OnOff);
}
```

Now we are also able to understand what happens in the `PositionServo` function.

```
double PositionServo(unsigned int ServoDigitalOutChannelNr, bool OnOff,
    double OnPulseDuration, double OffPulseDuration, double PulseSequenceDuration) {
  //HiTec HS-50 Servo specs: 50Hz, 0.9ms to 2.1ms, 1.5ms center. 0.09sec/60
  double PulseSeparation=20;
  if (!Output->InOutScaledNormalMode()) return PulseSequenceDuration;
  bool DirectOutMode=false;
  double Start=Output->GetTime();
  if (Output->IsInDirectOutputMode()) {
    DirectOutMode=true;
    Output->SetAssembleSequenceListMode();
    Output->StartSequence();
  } else Output->ChannelReservationList.CheckDigitalChannelReservation(
      ServoDigitalOutChannelNr, Start, PulseSequenceDuration);
  for (int i=0;i<(PulseSequenceDuration/PulseSeparation);i++) {
    Output->DigitalOutScaled(ServoDigitalOutChannelNr,On,On);
    Sequence.Wait((OnOff) ? OnPulseDuration : OffPulseDuration);
    Output->DigitalOutScaled(ServoDigitalOutChannelNr,Off,Off);
    Sequence.Wait((OnOff) ? (PulseSeparation-OnPulseDuration) :
                            (PulseSeparation-OffPulseDuration));
  }
  Sequence.GoToTime(Start,3455);
  if (DirectOutMode) {
    Sequence.Wait(PulseSequenceDuration+10);
    Sequence.StopSequence();
    Sequence.SetWaveformGenerationMode();
    Sequence.ExecuteSequenceList(/*ShowRunProgressDialog*/false);
    Sequence.EmptyNIcardFIFO();
  }
  return PulseSequenceDuration;
}
```

If the system is in direct output mode, the framing code around the generation of the servo motor pulse sequence brings the system into assemble sequence list mode, and executes that sequence at the end. This is required since servo motors require hight timing precision. In waveform generation mode the

```
Output->ChannelReservationList.CheckDigitalChannelReservation(
  ServoDigitalOutChannelNr, Start, PulseSequenceDuration);
```

command reserves the hardware digital channel of this servo motor for the time required for the motor action to occur. If two `PositionServo` commands reserve overlapping time windows on the same channel, an error message is generated. The `for` loop generates the pulse sequence required for the servo motor.

7.8 Loops

Sometimes you would like to program a sequence of commands that repeats till the user cancels it. The following programming example shows how to create a dialog box that displays some text, has a status bar and a cancel button. The code is executed till the dialog box is closed. Instead of the simple code switching the fiber MOT PIDs reference signal to 0 Volts and then back to its original value in direct output mode, you could also implement sophisticated sequences in waveform mode.

```
void CSequence::BlinkMOTFiber(CWnd* parent) {
  if ((CancelLoopDialog == NULL) && (parent)) {
    CancelLoopDialog = new CExecuteMeasurementDlg(parent,this);
    CancelLoopDialog->Create();
    CancelLoopDialog->SetWindowPos( &CWnd::wndTop ,100,200,150,150, SWP_NOZORDER
      | SWP_NOSIZE | SWP_DRAWFRAME );
  }
  int LoopNr=0;
  while ((CancelLoopDialog) && (LoopNr<10)) {
    LoopNr++;
    if (CancelLoopDialog) CancelLoopDialog->SetData("MOT fiber off",
      /*StatusBarAktValue*/0,/*StatusBarMaxValue*/1,/*PumpMessages*/true);
    else return;
    IOList.StoreAnalogOutValue("SetSrBlueMOTFiberReference");
    SetSrBlueMOTFiberReference(0);
    double TimeOver=0;
    while ((CancelLoopDialog) && (TimeOver<100)) {
      CancelLoopDialog->PumpMessages();
      Wait(1);
      TimeOver=TimeOver+1;
    }
    if (CancelLoopDialog) CancelLoopDialog->SetData("MOT fiber on",
      /*StatusBarAktValue*/1,/*StatusBarMaxValue*/1,/*PumpMessages*/true);
    IOList.RecallAnalogOutValue("SetSrRedMOTFiberReference");
    TimeOver=0;
    while ((CancelLoopDialog) && (TimeOver<100)) {
      CancelLoopDialog->PumpMessages();
      Wait(1);
      TimeOver=TimeOver+1;
    }
  }
  if (CancelLoopDialog) {
    CancelLoopDialog->DestroyWindow();
    CancelLoopDialog=NULL;
  }
}
```

The `CWnd* parent` parameter in the `BlinkMOTFiber` function call is required to link the new window to its parent. `BlinkMOTFiber` is called as explained in Sec. 6.4 in `CSequence::MessageMap` by

```
case IDM_BLINK_MOT_FIBER: BlinkMOTFiber(parent); break;
```

delivering this parameter. In this example the loop is executed at the most 10 times.

You can also create your own dialogs with other elements than text and status bar by duplicating `CExecuteMeasurementDlg`, renaming it to `CMyDlg` and modifying it so that it fits your needs. Use the resource editor to draw your dialog. You then need to add

```
#include "MyDlg.h"
static CMyDlg *MyCancelLoopDialog=NULL;
```

to `sequence.cpp`, `CancelLoopDialog=NULL;` to the constructor `CSequence::CSequence` and

```
if (MyCancelLoopDialog) {
  MyCancelLoopDialog->DestroyWindow();
  MyCancelLoopDialog=NULL;
}
```

to the destructor `CSequence::~CSequence` and

```
if (me==CancelLoopDialog) (CancelLoopDialog = NULL);
```

to `CSequence::ExecuteMeasurementDlgDone`. You can open the dialog and detect its status along the lines of the code example given above.

7.9 Idle and WakeUp function

If no action is performed by the program the CSequence::Idle(CWnd* parent) function is called. When buttons are pressed or an experimental sequence starts or stops the CSequence::WakeUp() function is called. This can be used to put sensitive or dangerous equipment like powerful lasers or power supplies in a save mode when no user is present and nothing is done with the experiment.

Here is the implementation of the idle function.

```
bool InIdle=false;

void CSequence::Idle(CWnd* parent) {
  if (InIdle) return;
  InIdle=true;
  bool CreateCancelDialog=!SaveMode;
  double TimeSinceLastBoot=GetSystemTime();//in seconds
  double ElapsedTime=TimeSinceLastBoot-LastWakeUpTime;
  if ((OvenShutterOffTime>0) && (ElapsedTime>OvenShutterOffTime) &&
      (!OvenShutterSaveMode)) {
    OvenShutterSaveMode=true;
    SaveMode=true;
    SwitchOvenShutter(Off);
    PlaySound("d:\\SrBEC\\ControlSrBEC\\Sound\\StarTrek\\C818.WAV",NULL,SND_FILENAME);
  }
  ... some more devices brought to save mode here
  if (SaveMode) {
    if (CreateCancelDialog && (IdleDialog == NULL) && (parent)) {
      IdleDialog = new CExecuteMeasurementDlg(parent,this);
      IdleDialog->Create();
      IdleDialog->SetWindowPos( &CWnd::wndTop ,100,200,150,150, SWP_NOZORDER |
                                SWP_NOSIZE | SWP_DRAWFRAME );
    }
    if (IdleDialog) {
      CString buf;
      buf.Format("Save mode activated\n\nOven shutter : %s\nPower supplies : %s",
        (OvenShutterSaveMode) ? "Off" : "On",(PowerSupplySaveMode) ? "Off" : "On");
      unsigned long ElapsedTimeInt=(unsigned long)ElapsedTime;
      IdleDialog->SetData(buf,ElapsedTimeInt%2,1,false);
      CheckDevices();
    } else WakeUp();
  }
  InIdle=false;
}
```

The variable `InIdle` assures it is only entered once. If no action occurred for longer than `OvenShutterOffTime` and this variable is ¿0, then the save mode is entered. The oven shutter is closed, a sound is played to alert the user that the save mode was entered. If no save mode dialog has been displayed before it is created now; see Sec. 7.8. Then the control is return to the caller of `CSequence::Idle` which will most likely call it right afterwards again in case the user has not done anything. The status bar blinks every second and perhaps some additional devices are brought into save mode later.

Sometimes it is also useful to not execute a task directly (because something timing critical needs to be done), but wait with it for some better time. A flag could be set instead of performing the task. This flag could be checked in `CSequence::Idle`. If it is set, the task is executed and the flag cleared.

As soon as the user clicks on any button or changes a menu `CSequence::WakeUp()` is called. It inhibits entering of the save mode, or exits the save mode.

```
void CSequence::WakeUp() {
  LastWakeUpTime=GetSystemTime();
  if (OvenShutterSaveMode) {
    OvenShutterSaveMode=false;
    SwitchOvenShutter(On);
  }
  ...  some more devices brought out of save mode here
  if (IdleDialog) {
    IdleDialog->DestroyWindow();
    IdleDialog=NULL;
  }
  SaveMode=false;
}
```

Chapter 8

Main experimental sequence

The main experimental sequence describes what should happen during the run of the experiment, typically starting with the loading of the magneto-optical trap and ending with some form of data acquisition like absorption imaging. It stands out from the other, simpler sequences in several ways. A button to start this sequence is integrated in every panel of the user interface. It is the sequence that will be called when taking automated sets of measurements. It provides some additional tools of organization that smaller sequences do not need.

Let us start with an overview. The "Run experiment" button on the user interface calls `CSequence::DoExperimentalSequence`. This procedure performs preparation work if necessary. The system might be initialized, the fluorescence trigger function might be prepared, some parameters of the sequence might be calculated and checked for consistency and validity. If everything is fine, the cameras are prepared for absorption imaging if required and finally the experimental sequence is prepared and executed as described in Sec. 7.2. After the sequence has been executed, the parameters are transferred to the data acquisition program, the experiment is reinitialized and the status of the machine is checked. All this will be described in more detail in Sec. 8.4.

The actual experimental sequence is programmed in `CSequence::ExperimentalSequence`. It starts with the `StartSequence` command, which this time contains a link to a function used to trigger the sequence when the MOT is sufficiently loaded. This trigger function is discussed in Sec. 8.3. Then the experimental sequence is described and finally stopped with the `StopSequence` command. To simplify the organization of the experimental sequence, it is divided into code blocks. The next section explains how to do that.

8.1 Experimental sequence code blocks

A typical experimental run of an ultracold atom machine involves a sequence of steps for example optical pumping, evaporative cooling, and absorption imaging. To organize this sequence of steps better, a certain programming style is used. This style is based on "code blocks". A code block describes one step of the experimental sequence, for example optical pumping, or one radiofrequency sweep, or absorption imaging. The user can select code blocks and quickly assemble an experimental sequence from them. One and the same code block can be used several times with different parameters. A code block requires the definition of some parameters on which it depends, among them one `bool` parameter that decides if the code block is used in a specific sequence or not. And it requires the actual code. The definition of a code block unifies parameter definition, registration and the sequence code using those parameters.

Let's create an example code block, one step in an evaporative cooling ramp. We will call this code block `RampOpticalDipoleTrap`. Every procedure name or variable name that has to do with this code block either contains `RampOpticalDipoleTrap` or the short name `RampOptDipTrap` to avoid creating parameters with the same name and to make the code more legible. The code block is placed in `sequence.cpp`. We show here the complete code block and discuss it below step by step.

```
//Ramp optical dipole trap
const unsigned int NrOptDipTrapRamps=10;
bool DoRampOpticalDipoleTrap[NrOptDipTrapRamps];
double RampDipTrapAttenuation[NrOptDipTrapRamps];
double RampDipTrapRampTime[NrOptDipTrapRamps];
double RampDipTrapRampFraction[NrOptDipTrapRamps];
double RampDipTrapWaitTime[NrOptDipTrapRamps];
void CSequence::RampOpticalDipoleTrap(int Nr) {
  if (!AssemblingParamList()) {
    if (!Decision("DoRampOpticalDipoleTrap"+itos(Nr))) return;
    StartNewWaveformGroup();
    Waveform(new CRamp("SetAttenuationDipoleTrapDDS"+itos(i),LastValue,
      RampDipTrapAttenuation[Nr],RampDipTrapRampTime[Nr],
      (RampDipTrapRampTime[Nr]>1000) ? 10 : 0.1));
    if (RampDipTrapRampFraction[Nr]!=100) {
      Wait(RampDipTrapRampTime[Nr]*0.01*RampDipTrapRampFraction[Nr]);
      RemoveWaveformGroup(GetCurrentWaveformGroupNumber());
    } else WaitTillEndOfWaveformGroup(GetCurrentWaveformGroupNumber());
    Wait(RampDipTrapWaitTime[Nr],1460);
  } else {
    if (Nr>=NrOptDipTrapRamps) {
      AfxMessageBox(
        "CParamList::AddRampOpticalDipoleTrap : too many optical dipole trap ramps.
         Increase NrOptDipTrapRamps.");
      return;
    }
    CString DDSUnits;
    DDSUnits.Format("%.0f..0dB",DDSAttenuationMax);
    ParamList.RegisterBool(&DoRampOpticalDipoleTrap[Nr],"DoRampOpticalDipoleTrap"+itos(Nr),
      "Ramp optical dipole trap "+itos(Nr)+" ?","D"+itos(Nr));
    ParamList.RegisterDouble(&RampDipTrapAttenuation[Nr], "RampDipTrapAttenuation"+itos(Nr),
      DDSAttenuationMax,0,"Attenuation",DDSUnits);
    ParamList.RegisterDouble(&RampDipTrapRampTime[Nr],"RampDipTrapRampTime"+itos(Nr),0,200000,
      "Ramp Time","ms");
    ParamList.RegisterDouble(&RampDipTrapRampFraction[Nr],"RampDipTrapRampFraction"+itos(Nr),
      0,100,"Ramp Fraction executed","%");
    ParamList.RegisterDouble(&RampDipTrapWaitTime[Nr],"RampDipTrapWaitTime"+itos(Nr),
      0.01,500000,"Wait Time at end of ramp","ms");
  }
}
```

The code block starts with the parameter definitions. `NrOptDipTrapRamps` defines how many code blocks of this type might maximally exist. The `bool` variable that starts with `Do` decides if a code block is executed. Since several code blocks of this type can exist, each parameter is now an array. The parameters defined here do not have to be declared external in `sequence.h` (or `paramlist.h`). The slight disadvantage of this simplification is that the parameters can only be used below their definition in `sequence.cpp`. If you need to use them also above, just put an external declaration into `sequence.h` or shuffle the different methods in `sequence.cpp` around until all usages of a parameter are below the definition of the parameter.

The method `CSequence::RampOpticalDipoleTrap(int Nr)` is called twice, once during the assembly of the `ParamList` and once when the experimental sequence is run. When it is called during `ParamList` assembly, `AssembleParamList()` delivers `true` and the registration part of `CSequence::RampOpticalDipoleTrap(int Nr)` is called. The `Register...` commands are the same as discussed in chapter 5, but you need to place `ParamList.` in front of them. You can also add `NewColumn`, `AddStatic` etc. commands, also adding `ParamList.` in front of them.

The second time `CSequence::RampOpticalDipoleTrap(int Nr)` is called is during the execution of the sequence and in this case `AssembleParamList()` delivers `false`. Now the actual sequence code is executed. The `Decision` command at the beginning is nearly equivalent to an `if (DoRampOpticalDipoleTrap[Nr])` statement. The only difference is that it adds the short description of the `DoRampOpticalDipoleTrap` variable defined when registering this variable to a special string in case the code block is executed. At the end of the experimental run this string contains a short description of the experimental sequence. If the sequences are short enough, this might be helpful when scanning through old measurements to quickly find out what was done. Note that the time step of the `CRamp` waveform is adapted in dependence of the total duration of the ramp. Note also the `Wait` command terminating the code block. The availability of a `Wait` command at the end of a code block is often useful.

To add this code block to the experimental sequence and at the same time to the parameter menus, you have to add a call to `CSequence::RampOpticalDipoleTrap` to `CSequence::MainExperimentalSequence()`. This method could look like the following:

```
void CSequence::MainExperimentalSequence() {
  CMOT();
  SwitchMOTOff();
  RampOpticalDipoleTrap(0);
  RampOpticalDipoleTrap(1);
  ParamList.NewMenu("Sequence parameters");
  FlashProbeBeam(0);
  LogWait();
  BlinkDipTrap(0);
  GoBackInTimeInSequence();
  ModulateDipTrap(0);
  ParamList.NewMenu("Detection parameters");
  RecaptureToMOT();
}
```

Note that the `RampOpticalDipoleTrap` method is called twice, with different `Nr` parameter. Care has to be taken that the `Nr` parameter is used in strictly increasing order starting from 0, which also tells you that a certain parameter may never be used twice for the same code block. This can be easily checked by searching for `RampOpticalDipoleTrap` in all files and analyzing the search result list.

The `CParamList::CParamList` constructor calls `Sequence.MainExperimentalSequence()`. Since during this call the system is in the assemble sequence list mode, `AssembleParamList()` delivers `true` and the registration commands of all code blocks are executed. When the experimental sequence is executed, `CSequence.ExperimentalSequence` calls `Sequence.MainExperimentalSequence()` and the sequence code of each code block is executed. In this way it is guaranteed that the parameter registration, and thus the parameter display on the user interface, happens in exactly the same order in which the code blocks are called during the experimental run.

8.2 Utilities

Utilities are small experimental sequences that perform simple tasks like testing a shutter. Programming of utilities is simplified by a scheme that follows the programming of code blocks closely. Here is an example:

```
//Utility DebugVision
CString *VisionDebugFilename;
bool CSequence::DebugVision(unsigned int Message, CWnd* parent) {
   if (!AssemblingUtilityDialog()) {
      switch (Message) {
         case IDM_VISION_DEBUG_START:
            Vision.DebugStart(*VisionDebugFilename);
         break;
         case IDM_VISION_DEBUG_STOP:
            Vision.DebugStop();
         break;
         default: return false;
      }
   } else {
      UtilityDialog.RegisterString(VisionDebugFilename,
         "VisionDebugFilename","c:\\FeLiKx\\VisionDebug.dat",
         200,"Vision debug Filename");
      UtilityDialog.AddButton(IDM_VISION_DEBUG_START,&Sequence);
      UtilityDialog.AddButton(IDM_VISION_DEBUG_STOP,&Sequence);
      UtilityDialog.AddStatic("");
   }
   return true;
}
```

Again the parameters on which the utility depends are defined above the method containing the code of the utility. And again this method contains two parts, the first containing the sequences that are executed if a certain button on the user interface is pressed and another part containing the registration of the parameters and declarations of the buttons. This time the registration commands have to be preceded by `UtilityDialog.` since the parameters are integrated into the utility dialog. When a button is pressed, the same code is called again, but this time `AssemblingUtilityDialog()` is false and the first part of the code is executed. The switch statement selects the code associated with a specific button. This code can contain sequences executed in waveform mode or loops. To be able to implement loops, the `CWnd* parent` parameter is available.

Each such utility needs to be called in `CSequence::MainUtilities` as follows.

```
bool CSequence::MainUtilities(unsigned int Message, CWnd* parent) {
   bool Received=false;
   if (AssemblingUtilityDialog()) UtilityDialog.NewMenu("Utilities");
   Received|=TestSequence(Message,parent);
   Received|=DebugVision(Message,parent);
   ...
   return Received;
}
```

Take a look at the other examples called there to get a good impression of what is possible.

8.3 Triggering the experimental sequence

It is often useful to start the experimental sequence only after trigger signals have been received. We usually use two trigger signals. First a software trigger activated when the MOT fluorescence has reached a certain level. After the software trigger has been given, the national instruments card is usually programmed to wait for an external hardware trigger. We use this to trigger the experimental sequence in a fixed phase relation to line. In combination with the clock signal that is phase locked to line, the whole experimental sequence is executed phase stable to line; see command `SyncToLine` in Sec. Sec:TimingCommands.

Here we will discuss the software trigger. The execution of the experimental sequence should start as fast as possible after the software trigger condition has been fulfilled. This requires that the table of values sent to the output cards has at least partially already been calculated and is ready for output. The software trigger function is called when either the table is full or when the sequence calculation has finished or enough of the table has been prepared (what enough means is determined in the "Systems configuration menu" by setting *Buffer Size For Start Waveform* to the desired duration in milliseconds). To enable the system to call the software trigger function, a pointer to the function is passed to it in the call of `StartSequence`. In addition a pointer to the parent window is passed to give the trigger function the possibility to display status dialogs.

```
StartSequence(&TriggerFunction,parent);
```

The trigger function is defined in the file `sequence.cpp` as follows:

```
bool TriggerFunction(CWnd* parent) {
  if (DebugOn) return true;
  bool OldDeleteWaveformAllowed=Output->DeleteWaveformAllowed;
  Output->SetDirectOutputMode();
  //Wait for something
  Sequence.PutAllChangedOutputsToStandardValueAfterFluorescenceTrigger();
  Output->Wait(1,1190);
  Output->SetWaveformGenerationMode();
  Output->DeleteWaveformAllowed=OldDeleteWaveformAllowed;
  return true;
}
```

The trigger function will usually analyze the value of one or more analog inputs and act on some outputs to bring the system into a desired state. After this state is reached the function returns. The trigger function is called during or at the end of the preparation of the output value table. This complicates things a bit. First the system is in waveform generation mode (but the waveform has not yet started). The `SetDirectOutputMode` brings the system temporarily back to direct output mode for this trigger function. Of course no command requiring waveform output can then be called in this function (e.g. no servo motor command, see Sec. 7.7). At the end the `SetWaveformGenerationMode` brings the system back to the waveform generation mode.

A second complication appears if the trigger function does not bring all outputs back to the same values they had at the start of the trigger function. In our lithium potassium mixture machine this is the case. First the trigger function waits till the Li MOT has reached a certain fluorescence, then it changes the magnetic fields to the values required for K loading and waits till the K MOT has reached a certain fluorescence. The reason for complications by this is that the calculation of the table of output values is based on a certain set of values that the outputs have before the waveform output is started. (We will call this set of values the values of the "output buffers" in the following.) This can enter for example into ramps if `LastValue` is given as start value for the ramp. Or it enters into the 16 data bits that have to be send to a digital output card, even if only one digital output is changed. If the fluorescence trigger function changes this set of initial values, the output table, which has already been partially calculated, is inconsistent with the new initial values and unwanted changes of the output channels could occur back to the values they had before the trigger function. To avoid this problem the function `PutAllChangedOutputsToStandardValueAfterFluorescenceTrigger` is used. Once it is called in a special mode at the beginning of the calculation of the output value table. In that mode the real hardware outputs are not changed, but the value of the output buffers is. So the output table is adjusted to the correct initial conditions after the trigger function. Two more calls are just for safety and could be left away. A second time this procedure is called at the end of the trigger function, just to make sure the outputs are really in these states. A third time it is called directly after the `StartSequence` command in `CSequence::ExperimentalSequence`. You have to place a call to each output that was modified in `PutAllChangedOutputsToStandardValueAfterFluorescenceTrigger` that sets these outputs to a well defined value. For a simple trigger function that reads only analog inputs this procedure could be

```
void CSequence::PutAllChangedOutputsToStandardValueAfterFluorescenceTrigger() {
  SwitchComparatorAnalogInSourceA0(On);
  SwitchComparatorAnalogInSourceA1(On);
  SwitchComparatorAnalogInSourceA2(On);
  SetComparatorAnalogInVoltage0(0);
}
```

Here is what the call to this procedure before waveform preparation looks like in
`CSequence::DoExperimentalSequence`:

```
Output->MakeOutBufferCopy();
Output->SetBlockOutputMode(true);
PutAllChangedOutputsToStandardValueAfterFluorescenceTrigger();
Wait(1,1240);
Output->SetBlockOutputMode(false);
```

The `SetBlockOutputMode` command blocks or unblocks access to the real hardware outputs. If the output is blocked, only the output buffers are changed. The `Wait` command makes sure all commands in `PutAllChangedOutputsToStandardValueAfterFluorescenceTrigger` are executed. `MakeOutBufferCopy` creates a copy of the output buffers. In the trigger function this copy is used as output buffers instead of the output buffers supporting the calculation of the output value table.

Take a look at the trigger function in the FeLiKx version of the control program. It demonstrates how to implement status dialogs within a trigger function. And it shows a method to guarantee the correct level of MOT fluorescence. If the initial MOT fluorescence is too high, the MOT is several times briefly switched off to kick out atoms until the MOT fluorescence is below the threshold value. Then the MOT is loaded again till the threshold is reached. If requested by the user, the K MOT is loaded in addition to the Li MOT.

8.4 The DoExperimentalSequence procedure

Let us take a look at `CSequence::DoExperimentalSequence`. This procedure is called to run the main experimental sequence. We will discuss some special things happening in this method.

```
WakeUp();
ResetSystemBeforeRun();
```

```
RunningExperimentalSequence=true;
if (!CheckParameterConsistency()) {
  RunningExperimentalSequence=false;
  return;
}
```

`WakeUp` is explained in section 7.9. `ResetSystemBeforeRun` will call `ResetSystem` in case the user has requested to do so. `CheckParameterConsistency` can contain checks of parameter consistency. It should display an error message using `AfxMessageBox` in case there is a problem. If it returns `false` the sequence is not executed. The flag `RunningExperimentalSequence` is set if the experiment is run.

```
CalculateDependentParameters();
```

can adapt parameters depending on others. For example the frequency of rf transitions can be adapted to current values through magnetic field coils.

```
Output->DebugSync(DebugSyncOn,*DebugSyncFileName);
```

A table of all GPIB, serial port device or input port commands is generated after the run of the sequence for debugging purposes if `DebugSyncOn` is `true`.

```
Output->MakeOutBufferCopy();
Output->SetBlockOutputMode(true);
PutAllChangedOutputsToStandardValueAfterFluorescenceTrigger();
Wait(1,1240);
Output->SetBlockOutputMode(false);
```

This has been discussed above. Next comes the assembly and execution of the sequence list.

```
SetAssembleSequenceListMode();
ExperimentalSequence(parent);
if (!InitializeSequence()) {
  SetDirectOutputMode();
  Output->EmptySequenceList();
  RunningExperimentalSequence=false;
  return;
}
SetWaveformGenerationMode();
ExecuteSequenceList();
if (DebugSequenceListOn) DebugSequenceList("D:\\SequenceListAfterExecution.dat");
EmptyNIcardFIFO();
```

`InitializeSequence()` prepares the cameras for imaging. It does not send the software trigger command for imaging to the cameras. This is only done in the software trigger function at an appropriate time. If something goes wrong, the sequence list is emptied. If everything goes well, the sequence is executed. `ExecuteSequenceList` calls the software trigger function after enough of the output table has been calculated. `DebugSequenceList` stores a table of the sequence list in a file for debugging purposes.

The next lines of code reinitialize the system.

```
SwitchForceWritingMode(On);
SetAssembleSequenceListMode();
```

```
StartSequence();
InitializeSystem(/*UseSlowRamps*/false);
Wait(171);
StopSequence();
SetWaveformGenerationMode();
SwitchForceWritingMode(On);
ExecuteSequenceList();
EmptyNIcardFIFO();
SwitchForceWritingMode(Off);
Wait(10,1260);
StartLoadingTime=GetTickCount();
```

The "force writing mode" assures that every output command is written, even if the output buffers indicate that the output is already in the state that should be written. This is just for safety and probably unnecessary. `StartLoadingTime` contains the last time MOT loading was started.

Next comes a chunk of code that reads analog inputs or the oven temperature if requested. These values are then transferred to the data acquisition program. Since the code is lengthy, but simple it is not reproduced here. The parameters of this experimental run are transferred to the data acquisition program using `Vision.SendPictureData` and `Vision.SendDataFile`. At the end it is checked that everything did work fine, especially that the MOT did load. If it did not do that for a few consecutive runs, an error dialog is displayed giving the user the chance to relock the lasers and continue a set of measurements without taking too many useless data points.

8.5 Communication with the data acquisition program "Vision"

The control program communicates with the data acquisition program Vision using TCP/IP. In `InitializeSequence()` (which is called in `CSequence::DoExperimentalSequence`) it is checked that the cameras are ready using `Vision.CheckReady()`, then the camera parameters are sent using `Vision.SetNetCameraParameters`. During the trigger function, the camera software trigger is sent to Vision using `Vision.TakeAbsorptionPicture`. Vision passes this signal on to the camera computers. After the execution of the experimental sequence `Vision.Ready();` waits till the pictures have been acquired. Then additional data is sent using `Vision.SendPictureData`. This includes all parameters of the experimental run, which Vision stores as text file together with the images. Additional text files can be saved using `Vision.SendDataFile` on the data acquisition computer. `Vision.RunFinished` tells Vision that the run has finished. If sets of experiments are performed in an automated manner, some additional information is sent to Vision. This will be not explained here. You can see how this works in `CSequenceLib::ExecuteMeasurement`.

Chapter 9

User interface

If you do not customize the user interface by modifying the `CMainDialog` class, it will look somewhat like this:

```
Manual operations:
>MOT lasers...
>power supplies...
>dipole trap...

Parameter menus:
>Initial parameters...
>Sequence parameters...
>Detection parameters...

>Configuration parameters...
>General information...

Actions:
>Run experiment
>Queue experiment
>Reference queue experiment

>Measurements
>Measurement queue
>Reference Measurement queue

>Utilities
```

">" signifies a button and "..." that a sub dialog will pop up if the button is pressed.

You can see the values of the outputs in the first two menus, the relation between the output name and the channel number and you can modify the value written to the outputs. Right-click anywhere or press the *Apply* or *Ok* buttons to update. Click *Cancel* to leave the menus without updating the values. Scroll the mouse wheel up or down to change between menus.

The next menus permit to change the parameters.

The configuration parameters contain the IP address to Vision, and the debug modes. General information shows what happened during the last experimental run, eg the TCP/IP communication with Vision or the duration of the last run.

The "Run experiment" button runs one experiment. "Queue experiment" sets such a run into the measurement queue.

9.1 Set of measurements

"Measurements" show a panel of slots for user defined types of measurements. By selecting a slot, you will be shown a parameter panel, where you can select which parameter to vary in which way to perform the measurement. Eg the expansion time of an atomic cloud for a time of flight temperature measurement. You can execute the so defined measurement or you can put it into the measurement queue.

9.2 Measurement queue

"Measurement queue" shows the contents of the queue and provides buttons to start the queue, iterate through it or clear its contents.

Utilities shows user defined functions and some standard utilities like, saving and loading the parameters into files with arbitrary names, calibrating a VCO (a HP5334A counter has to be hooked up on address 24 of the GPIB bus) or calibrating the MOT fluorescence (for this, some analog input function has to be defined and called in the MeasureFluorescence method of CSequence).

9.3 Cyclic operation

Cyclic operation improves the stability of the system. The experiment runs one experimental sequence after the other without pause. If the user does not specify new parameters, the old ones are used several times. To enter cyclic operation, press the *Start cyclic operation* button. Then call the control program a second time, e.g. by calling the control.exe file directly instead of using the Visual Studio development studio. The second instance of the program will detect the presence of the first and run purely as user interface. Parameters can be modified and transferred to the instance of control that runs the experiment by pressing the *Run Experiment* button. A dialog window shows up that displays the number of the experimental run that will be the first that uses the new parameters. Also experimental series can be sent to the cycling program using the measurement menu. `Stop cyclic operation` sends a command interrupting the cyclic operation.

The communication between the two instances of control is simply performed by accessing files on the hard disk. The filenames are defined in `SequenceLib.cpp` as

```
CString CycleBaseFileName="c:\\CycleCommand%u.dat";
CString CycleSemaphoreFileName="c:\\CycleSemaphore.dat";
CString RemoteSemaphoreFileName="c:\\CycleSemaphoreRemote.dat";
CString CyclePictureNumberFileName="c:\\CyclePicNr.dat";
```

9.4 The system configuration menu

This section needs to be updated

The system configuration menu contains parameters needed by every implementation of the control program, independent of the exact experimental sequence, some system results and some system utilities.

Here is an explanation of the parameters.

- ExternalTrigger: triggered through master timer port hardware trigger input or soft trigger.
- ExternalClock: master timer internal or external clock.
- ConnectToVision: connect to vision using VisionComputerIPAdress and VisionPort.
- DebugOn: debug on. Write in DebugFileName for timesteps with DebugDeltaTime spacing. DebugOriginShift: list nicely as multi channel oscilloscope in Origin. Output in MHZ, I, etc: physical units output, not compatible with GoBackInTime() command or DebugOriginShift. DebugAnalogGain: just to make low voltages better visible
- DebugSyncOn: produces list of GPIB, serial port etc. commands and info about their execution, writes to DebugSyncFileName.
- DebugWaveformOn: create list of waveform commands, stores in DebugWaveformFileName.
- DoTimingJitterCompensation: stabilizes timedelay for consecutive execution of sequences in a measurement series. TimingJitterOffsetDelay: added security time.
- ShowRunProgressDialog: shows a dialog displaying fill size of NI buffer, total time in buffer and total executed time during the execution of the sequence.
- BufferSizeForStartWaveform: The waveform starts when the buffer has been filled up to this time.

The results that are displayed are the JitterCompensationDelay, the LastErrorMessage and the contents of the last 10 TCP/IP communications.

Chapter 10

Debugging

10.1 Sequence debugging

10.1.1 Sequence list

A list containing all commands of the experimental sequence can be written into a file after each sequence by enabling that option in the system configuration menu. Three files are created. One with a human readable list, one with a computer readable list and one computer readable list of the initial states of all outputs. The two computer readable lists can be displayed graphically using the Matlab program available on the control system webpage (www.nintaka.com). This program has been written by Christoph Kohstall.

10.1.2 MultiIO dump

You can write the commands written to the MultiIO bus into an ASCII file using the "Debug MultiIO" option in the system configuration menu. Usually this option should be switched off, since it can consume a lot of time if the sequence is long. Simple sequences useful for debugging are created for example when leaving a manual control menu by pressing the "Ok" button (or left click with the mouse into a gray area).

10.2 Hardware debugging

To be efficient at debugging the MultiIO hardware, it is highly recommended to set up a testbed for this hardware, independent of the main experiment. This allows to check all components independently. If something breaks on the main machine, the error can easily be localized by swapping modules that are under suspicion of being broken by modules that have been proven to work in the testbed. This concerns bus driver modules, including sub bus decoder modules and DDS driver modules, all kinds of output cards and the cables connecting these modules.

In addition some devices are useful debugging tools, for example a circuit board that breaks the bus system signals out to individual BNC connectors and LEDs indicating the status.

10.2.1 Checking address decoding

A source of error for devices on the MultiIO bus system is incomplete address decoding. The reason for this error are disconnected address lines because of damaged cables or bad soldering spots. To find this type of error, random data and address signals are written to the bus, excluding the address that should be tested. If address decoding is incomplete, some signals will go through and this can be observed on an oscilloscope. The procedure used to check devices for incomplete address decoding is called by the button "Multi IO shuffle test" in the utility menu. The parameters for this test are the following. `Excluded Bus` and `Excluded Address` specify a subbus and an address to which no signals should be sent. If `Test strobe box` is set, then all signals to the specified subbus are blocked, otherwise only signals for which subbus and address match. `Address Mask` and `Address Pattern` specify a mask that is applied to the address in binary format. If the mask contains 1, the bit specified is randomly toggled, if it contains 0, the value from the specified pattern is used. The same applies to `Data Mask` and `Data Pattern` if `Mask Data If Address Matched` is set. Using these options you can find out which address line is interrupted.

10.2.2 Checking a digital output card

In the system parameters menu you can specify the software address of a digital output. Connect LEDs to each output (don't forget 400 Ohm current limiting resistors in series with each LED). The button "Test digital out 16bit" displays a pattern on the 16 outputs starting with the output specified. The button "Test digital out" blinks one output.

10.2.3 Checking the digital inputs

The button "Test digital input" reads and displays the status of each digital input.

10.2.4 Checking an analog input card

The button "Analog in recorder" reads, displays and stores in a file the value of the analog inputs of the input card specified.

10.2.5 Checking an analog output card

The button "Test analog out" writes a sawtooth signal on the analog output specified.

An analog output card can be thoroughly tested by connecting the eight outputs to the eight inputs of an analog input card. The button "Test analog out card" runs a complete test on the output board specified and stores the results in an ASCII file.

10.2.6 Checking a DDS

The button "Test DDS" allows to test the DDS with the specified software address. Connect the DDS to a spectrum analyzer. The frequency is swept slowly from 0 to 135 MHz, then the intensity is reduced and increased and finally the DDS is set to 100 MHz at maximum power.

Chapter 11

Simple extensions

11.1 Different parallel out card for MultiIO system

If you want to use a different digital output card than the NI6533 card for the MultiIO bus system, then modify the `CNI653x` class accordingly. Not many functions are used to communicate with this class. We just need to give the memory address of the ring buffer to the device driver of the new card, start the data transmission and check at which point within the data transmission we are.

11.2 Connecting new types of devices to the MultiIO bus system

You might want to connect new types of devices to the MultiIO bus system, for example other DDS types or microcontrollers. To do so, you need to derive a class from `CMultiWriteDevice`. To do so, duplicate the `CAD9858` class and modify it to your needs. It contains a mechanism to store lists of commands, that are subsequently written to the bus in several cycles. Next you need to duplicate the

```
CAD9858 *CMultiIO::AddAD9858(unsigned short aBus, unsigned int aBaseAddress,
  double externalClockSpeed, double FrequencyMultiplier)
```

function in `IOList.cpp` and adjust it to your needs. An instance of your new class has to be created, added to the `MultiWriteDeviceList` list and the `MultiWriteDeviceTable` table and, if you want, to your own table numbering your new device as the `AD9858` table in the example. The function you created in this way will be called in `CSequence::ConfigureHardware()` to add a new device to the system.

Next you have to duplicate and adapt the

```
void COutput::SetAD9858Value(unsigned int DDSNr, double UnscaledValue,
  double dValue, int Type);
```

method of `COutput`. It serves to link registers internal to your device to virtual analog and digital output ports that can be influenced using the manual control menus. Don't forget to change `TypeOfMultiWriteDevice` to a new and unique value in

```
SequenceList.Add(new CSequenceListMultiWriteDeviceValue(/*TypeOfMultiWriteDevice*/1,
  AD9858MultiWriteDeviceNr[DDSNr],DDSNr,UnscaledValue,dValue,Type));
```

Add your new `SetXXXValue` function in `CSequenceListMultiWriteDeviceValue::ExecutePoint()`.

Finally you have to add procedures like

```
SetFrequencyDDSAD9858(unsigned int DDSNumber, double Frequency);
```

to `IOList.h` and `IOList.cpp`. These functions are called in the functions that give names to your outputs and that are registered in the constructor of `CIOList`. You can figure out how this works by analyzing how `AD9858` calls are handled.