QDSP
Domain-Specific Language

User Manual:

Education and semester:

Robot Systems Engineering - 9th semester

Revision:

1.0 - MMMI SDU 01/2014

Supervisors:

Anders Stengaard Sørensen & Ulrik Pagh Schultz

Developed by:

Jon Henneberg & Anders Krauthammer

Faculty of Engineering
University of Southern Denmark

Contents

Getting started 1

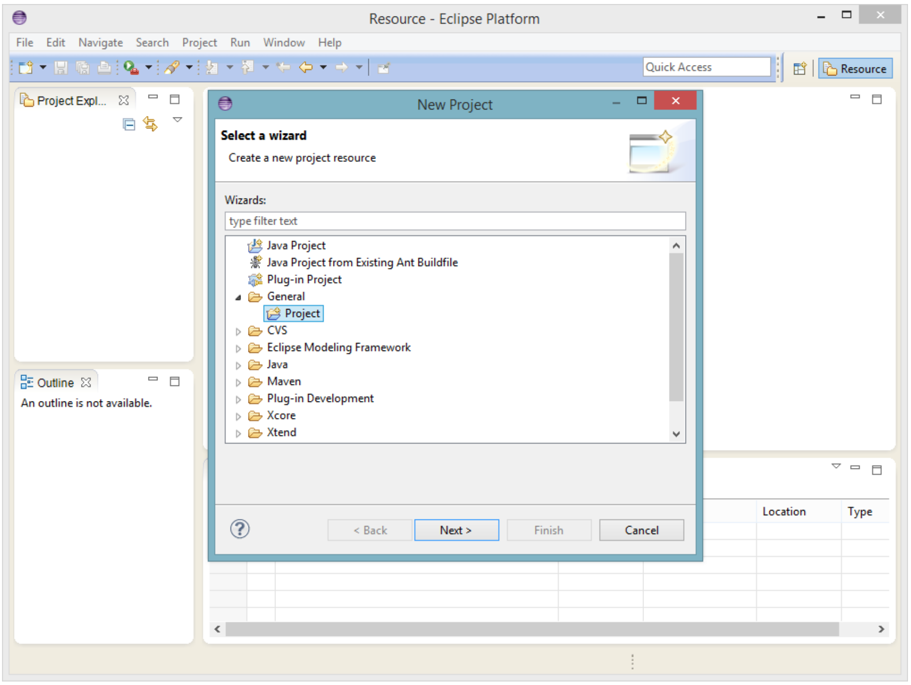
1.1 Installing QDSP DSL

The QDSP language is a external domain specific language, provided as an eclipse plugin. To install the plugin in an existing eclipse through the “Intsall new software...” under the Help menu, using the eclipse update site listed in the end of this section. Note that the plug in is written to Eclipse Platform Version: 4.3.0 and are not tested on other versions of eclipse.

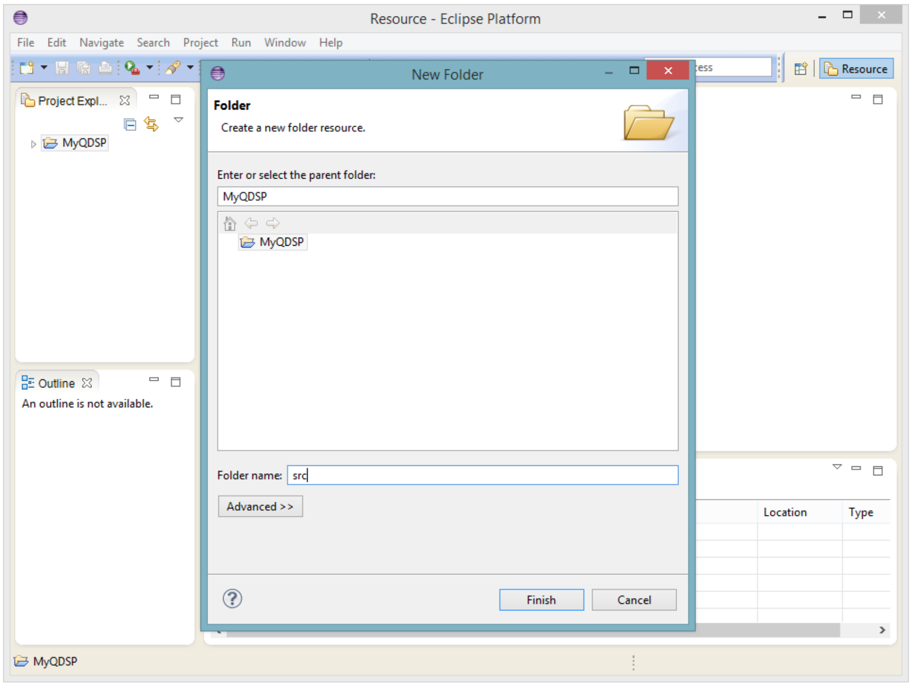
Eclipse update site: `https://raw.github.com/QDSP/installsite/master`

1.2 Creating a QDSP project

When eclipse is downloaded, and the plugin for the QDSP DSL is installed, you are ready to create the project. Open eclipse and choose your workspace.



- Go to File -> New and choose project
- Choose Project under General and click Next
- Enter a project title and click Finish


Resource - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Project Expl...
Outline
Resource
Location
Type
Advanced >>
Finish
Cancel

New Folder

Folder

Create a new folder resource.

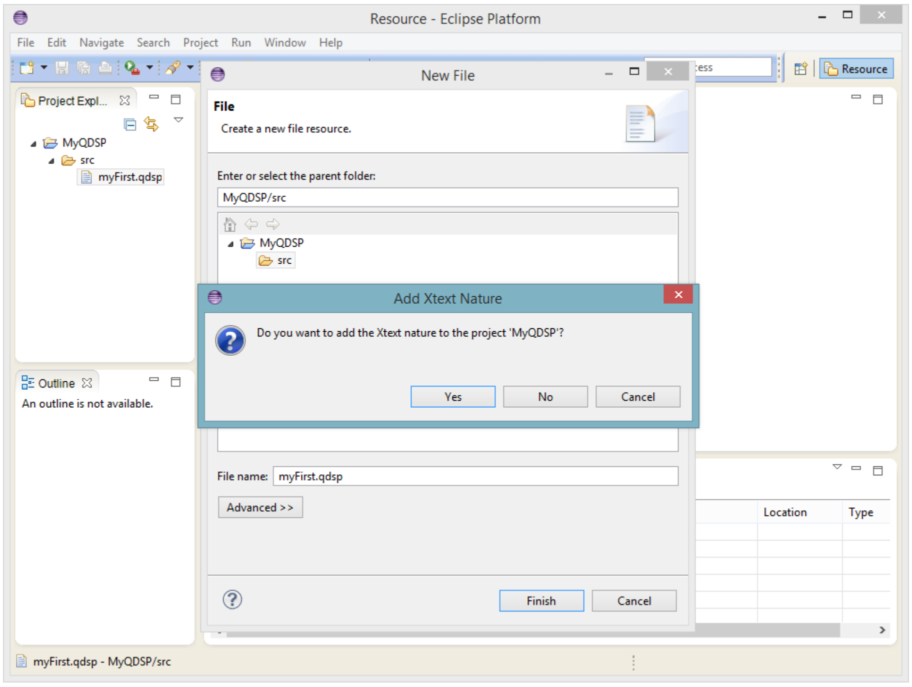
Enter or select the parent folder: MyQDSP

Folder name: src

An outline is not available.

MyQDSP

- Rightclick project -> New and choose Folder
- Set folder name to “scr” and click Finish


Resource - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Project Expl...
Outline
Resource
Location
Type
Yes
No
Cancel
Advanced >>
Finish
Cancel

New File

File

Create a new file resource.

Enter or select the parent folder: MyQDSP/src

Add Xtext Nature

Do you want to add the Xtext nature to the project 'MyQDSP'?

File name: myFirst.qdsp

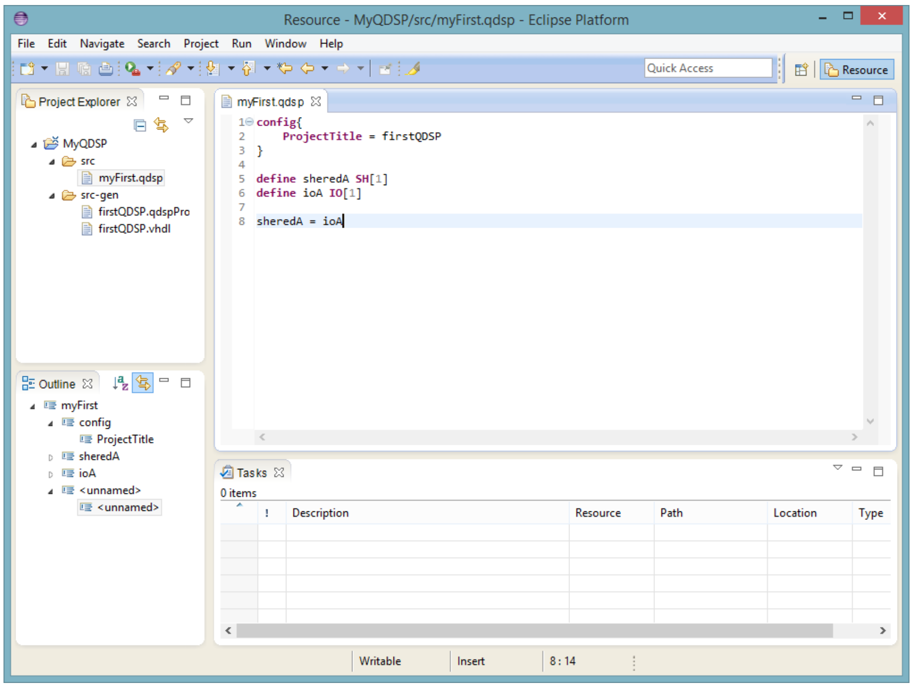
An outline is not available.

myFirst.qdsp - MyQDSP/src

- Rightclick scr -> New and choose File
- Enter file name and click Finish
- Click Yes in the “Add Xtext Nature” pop up

1.3 Generated output

The QDSP DSL compiler generates at least two files, a “.qdspPro” file containing the raw micro code for the written program, and a “.vhdl” file with a version of the QDSP3 optimized for the given micro code. This means that the size of heap and the program memory is as small as possible, and the instruction set is trimmed to only contain the used instructions. The QDSP3 in the “.vhdl” file is already containing the microcode and the default values for the heap, and are therefor ready to use in a Xilinx ISE project. Just add the file to the project, and instantiate the module. All generated files can be found in the “scr-gen” folder.



```
config{
    ProjectTitle = firstQDSP
}

define sheredA SH[1]
define ioA IO[1]

sheredA = ioA
```

If the run-time programming is enabled the size of heap and the program memory is set to the values defined in the configuration. It also generates two more files. A “.bin” file containing a header telling the needed resources for the program to run, the micro code, and the default values for the heap. All of this is stored in a binary format. The last file “qdspProgrammer.py” is a Python script used to program the QDSP through Unity-Link or μTosNet. The programmer is specified to the configuration of the current QDSP.

QDSP 2

2.1 Concept

The basis of the design comes from the QDSP designed by Anders Steengaard. This design was maintained in the VHDL as to support interoperability between the old and new design while additional features were added. The overall design of the DSP can be seen in figure 2.1.1. The design was created to allow the DSP to be inserted in applications that use μTosNet or UnityLink and only requre minimal changes to the existing VHDL. The result of this design is that the DSP has two main memory areas, on to interface with the surrounding vhdl, called the IO memory, and one to either μTosNet or Unity-Link refereed to as the Shared memory.

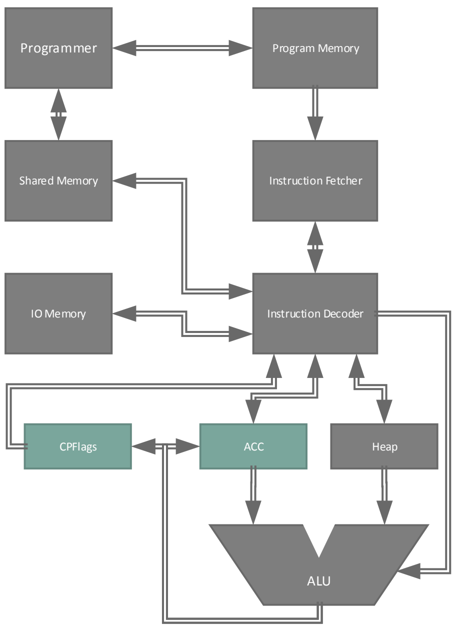


Figure 2.1.1: Block diagram of QDSP

2.2 Instructions

The structure of the instruction-set from the original QDSP was designed to allow for arguments to be provided along with the instruction. This was done by having a fixed length instruction followed by argument, this structure was maintained in this new design.

The instruction-set was updated to allow for new features such as branching whereby evaluations on values could be made and subsequent jumps in the program code could be made, allowing for loops and conditional code execution.

The full instruction set available for the QDSP is seen in table 2.2.1

Name	Argument	Description
NOP	N/A	No Operation
CAH	Heap address	Copy accumulator to heap
CHA	Heap address	Copy heap value to accumulator
RST	N/A	Reset program counter to 0
LDX	N/A	Test if program loader is requested on memory interface
CAS	Memory address	Copy accumulator to shared memory
CSA	Memory address	Copy shared memory value to accumulator
CAI	Memory address	Copy accumulator to IO memory
CIA	Memory address	Copy IO memory value to accumulator
ADD	Heap address	Add Heap to accumulator
SUB	Heap address	Subtract heap from Accumulator
BDIV	Shift count	Signed binary division using bit shifting
BMUL	Shift count	Signed binary multiplication using bit shifting
MUL	Heap address	Multiply Heap with accumulator
TRN	N/A	Truncate accumulator to 16bit value
MAX	Heap address	Compares accumulator to heap value and set accumulator to the highest of the two
MIN	Heap address	Compares accumulator to heap value and set accumulator to the smallest of the two
LSH	Shift count	Left shifts accumulator the given amount
RSH	Shift count	Right shifts accumulator the given amount
JMP	Program memory address	Changes program counter to specified program memory address
BRE	Program memory address	Changes program counter to specified program memory address, based on compare flags
BRG	Program memory address	Changes program counter to specified program memory address, based on compare flags
BRL	Program memory address	Changes program counter to specified program memory address, based on compare flags
CMP	Heap address	Compares heap value with accumulator and set compare flags based on the results

Table 2.2.1: QDSP Instruction set

QDSP Programing language 3

3.1 Language structure

The QDSP language, also called QDSP DSL, is designed according to the syntax from the language C. The code structure is based around a few different components, that can be added to the code. The main component is assignment operations, see section 3.1.5, that can assign math and data formats to eg. variables or IO memory.

It is also possible to define variables, constants, and addresses, for easier code writing see section 3.1.3. Note that it is only possible to define this in the main level of the code, and not inside the predefined functions. It is not possible to create functions in the QDSP DSL, but a few predefined functions is available see section 3.1.2 and 3.2.

Example 3.1.1:

```
define  input IO[1]
define output SH[1]

output = input
```

The micro code created by the compiler, loops around at the end of the code, so the output of example 3.1.1 and example 3.1.2 will give the same result. The benefit of adding the `while(true)` is to make initial calculations ore like before the `while(true)`, that only will be executed once.

Example 3.1.2:

```
define  input IO[1]
define output SH[1]

while(true){
  output = input
  // more code
}
```

At last it is possible to add a configuration block. It is only possible to have one configuration block, and it have to be the first part of the code. More about the configuration block can be found in section 3.1.1.

3.1.1 Configurations

It is possible to add a configuration block as the firs part of the QODS code. This allows the programmer to change some parameters used by the compiler. The most of the parameters, are only used then the QDSP is set to be run time programmable. An example on how to add a configuration block can be found in example 3.1.3, and the full list of parameters are listed in table 3.1.1

Example 3.1.3:

```
config{
  ProjectTitle = robotrainer
  EnableOptimization = false
  SHInterfaceType = Unity-Link
}
```

Name	Type	Default	Description
ProjectTitle	String	NewQdsp	Title of the project
AutoCommentQDSPCode	Boolean	true	Comments the micro instructions
EnableOptimization	Boolean	true	Assigns the programing transmit address on SH memory for run time programming
RuntimeProgrammable	Boolean	false	Enables the capability of run time program the QDSP
HeapSize[1]	Integer	32	The amount of Heap registers.
ProgramMemorySize[1]	Integer	128	The amount of program memory available.
ProgramingControlAddress[1]	Integer	7	Assigns the programing control address on SH memory for run time programming
ProgramingStatusAddress[1]	Integer	0	Assigns the programing status address on SH memory for run time programming
ProgramingDataInAddress[1]	Integer	1	Assigns the programing revise address on SH memory for run time programming
ProgramingDataOutAddress[1]	Integer	6	
SHInterfaceType[1]	String	Unity-Link	Changes the SH interface to match Unity-Link or uTosNet

Table 3.1.1: Configurations

3.1.2 Functions

There are to different kinds of functions, void functions and math functions. Void functions, like `sleep()` or `if()`, are used to tell the compiler to implement some code, or how to handle a part of the code. And Math functions are functions like `max()` or `parenthesis()`. These functions can only be used doing math operations, they leaves the result on the accumulator. For a full list of functions, and the syntax, see section 3.2

3.1.3 Data formats

The QDSP DSL allow the programmer to use a few simple data formats, which make it a lot easier write code, and the code becomes more descriptive. The available formats are explained further beneath, and an example of how to define and use them is given.

Address

The term address in the QDSP DSL is used to access the IO and shared memory, to exchange values with ether the computer connected through Unity-Link or μTosNet, or with the rest of the FPGA. Example 3.1.4 shows how to pipe IO memory address 1 through the QDSP to shared memory address 1, without doing anything with the data.

Example 3.1.4:

```
SH[1] = IO[1]
```

[1]Only used if RuntimeProgrammable it true

Define

Defines are used to name addresses which makes the code more descriptive. Example 3.1.5 again shows the the piping of IO memory address 1 to shared memory address 1 as example 3.1.4, but now the addresses is defined to names.

Example 3.1.5:

```
define sheredA SH[1]
define ioA IO[1]

sheredA = ioA
```

Variable

Defining a variable reserves a heap register which afterwards can be referred to by the given name. It is optional to directly assign a value to the variable during the definition. A variable is a signed 32 bit value.

Example 3.1.6:

```
var var1 = 5
var var2

var2 = var1
```

Constant

Constants is a lot like the variables section 3.1.3. The only differences is that the constant is read only, and have to be assigned to a value. The definition and use can be seen in example 3.1.7.

Example 3.1.7:

```
const offset = 256

SH[1] = IO[1] + offset
```

Number

The QDSP is a 32 bit processor, and uses a 32 bit signed datatype for variables, constants, and numbers. Therefor the usable numbers are in the range from $-2{,}147{,}483{,}648$ to $2{,}147{,}483{,}647$.

3.1.4 Boolean expression

Boolean expressions are used in some of the available functions to determine if a given code have to be executed. In the QDSP DSL an Boolean expression can be a simple `true` or `false`, or it can be a comparison between to *Math operation* using one of the comparators from table 3.1.2.

Example 3.1.8:

```
if(true){
  //code
}
```

Example 3.1.9:

```
var x = SH[1]
var y = SH[2]
if(x + 5 >= y * 2){
  //code
}
```

Comparator	Description
==	Equal
!=	Not equal
<	Less than
<=	Less than or equal
>	Grater than
>=	Grater than or equal

Table 3.1.2: Math comparators

3.1.5 Assignment operation

An assignment operation is a writable data format such as variables, defines, and addresses, followed by one of the assignment operators listed in table 3.1.3. The most of the assignment operators takes an argument, in form of a math operation or one of the data formats. For information about math operations see section 3.1.6 or data formats see section 3.1.3.

Example 3.1.10:

```
define in1  IO[1]
define in2  IO[1]
define out1 IO[4]
var var1
const const1 = 10

var1 = 5
var1 += const1 + in1 - in2
var1 ++
out1 = var1
```

Operator	Description
=	Sets the data to the argument
+=	Adds the argument
-=	Subtracts the argument
*=	Multiplys with the argument
++	Adds 1
--	Subtracts 1

Table 3.1.3: Assignment operators

3.1.6 Math operation

The math operation, is a series of data formats like *numbers*, *variables*, *constants*, or *defines* separate by a math operator. The different operators can be seen in table 3.1.4. Note that the the bit shift operators only can be followed by a *number* telling the amount of bit shifts to be executed.

Example 3.1.11:

```
define  input IO[1]

input + 5 * 2
```

Please note that compiler does not comply to the standard order of operations rules. The compiler solves the math from left to right. This means that the result of example 3.1.11 will be 20 and not 15 for an input of 5. It is possible to force the the order by using parentheses as done in example 3.1.12.

Example 3.1.12:

```
define  input SH[1]

input + (5 * 2)
```

It is worth to mention that adding five times two in the way it is done in example 3.1.12 is bad, taking into account that the compiler does not do any precompiling, and therefore this will result in a micro code that makes the QDSP actually multiplicity tow and five. The better way would be to just add ten.

Operator	Description
+	Add
-	Subtract
*	Multiply[2]
/	Divide[3]
«	Left bit shift
»	Right bit shift

Table 3.1.4: Math operators

3.2 Functions

It is not possible to create your own functions, but the programing language consists of a small amount of predefined functions, that can be used doing the programming. Words written in Italic, is a reference to a data format, see section 3.1.3

3.2.1 sleep

`void sleep(CPUclocks)`

The sleep function creates a number of NOP operations in the micro code, stalling the processor for the amount of clocks given. It is not a good choice for long sleeps, because every NOP operation added, uses space in the program memory. It is often better to make a loop using a counter variable.

Takes: A *number* of clocks to sleep.

[2]The multiplier multiplies to values and stores the result in a 32 bit value make sure that the result dos not exceeds a 32 bit value.
[3]It is only possible to divide by a number that is a power of 2. It is therefore not possible to divide by variables and constants.

3.2.2 runBootLoader

`void runBootLoader()`

The runBootLoader function is only used then the QDSP is set to be run time programmable see section 3.3. This gives the programmer the ability to determine then the QDSP will run the few instructions making it possible to stop and program the QDSP from a computer. It is important that this function is called frequently for the run time functionality to work. it is therefore important to add the `runBootLoader()` command inside the `while(true)` in the case seen in example 3.1.2.

3.2.3 while

`void while(`*`boolean`*`){ code }`

The while function creates jumps and branches in the micro code, with the result that the given code runs as long as the expression it true.

Example 3.2.1:

```
while(true){
  //code
}
```

Example 3.2.2:

```
var i = 0
while(i < 10){
  //code
  i++
}
```

3.2.4 do while

`void do{ code } while(`*`boolean`*`)`

The do while function creates jumps and branches in the micro code, to ensuring that the given code is running at least one time, and continues as long as the expression it true.

Example 3.2.3:

```
var i = 0
do{
  //code
  i++
} while (i < 10)
```

3.2.5 for

`void for(`*`Assignment operation ; boolean ; Assignment operation`*`){ code }`

The while function executes the first assignment operation. Afterward jumps and branches are created in the micro code, so the given code is running as long as the expression it true. The last assignment operation is added to the code.

Example 3.2.4:

```
var i
for(i = 0 ; i < 10 ; i++){
  //code
}
```

3.2.6 if

`void if(`*boolean*`){ code } else if(`*boolean*`){ code } else { code }`

The if function creates jumps and branches in the micro code, to obtain that the given code is running if the expression it true. The number of else if expressions is up to the programmer, and the else expression is optional.

Example 3.2.5:

```
var v = IO[1]
if(v == 1){
  //code
} else if(v >= 2){
  //code
} else {
  //code
}
```

3.2.7 switch case

`void switch` *Math operation* `{ case` *number* `or` *constant* `:  code }`

The switch case function, executes the *Math operation* and runs the code from the specified case with the right result. If a case with the right result does not exist, the default is used. To stop the code at the end of a case, the break command is used.

Example 3.2.6:

```
var v = IO[1]
switch v {
  case 1:
    //code
    break
  case 2:
    //code
  default:
    //code
}
```

3.2.8 trunctate16

`trunctate16(`*Math operation*`)`

The trunctate16 function, executes the *Math operation*, and fits the result into a 16 bit value. If the result is grater than the max value of a 16 bit number (65535) the output of this function will be 65535, else the output will be the result of the *Math operation*.

3.2.9 min

`min(`*Math operation* `,` *constant* `or` *variable* `or` *number*`)`

The min function will return the minimum value of the two inputs.

3.2.10 max

`max(`*Math operation* `,` *constant* `or` *variable* `or` *number*`)`

The max function will return the maximum value of the two inputs.

3.3 Run time programming the QDSP

The run time programming functionality, is designed to utilize the existing computer interface of the QDSP, was added to allow for the run-time programming the program memory, allowing for faster iterations on the code running on the DSP. The run time programming functionality is by default disabled, but can be enabled in the configuration.

The QDSP can be programed in two ways, using the Python programing script generated by the compiler, or by using the programing interface.

3.3.1 Python programing script

The python script requires Python3 and PySerial[4] in order to run. The script takes three arguments, the file to send, the port and the baud rate. The latter two are optional as the script asks for a port if none is given and uses a default baud rate.

The script can be called from within Eclipse by creating an external tool configuration and adding a program, by using the following parameters:

Location:	location of python
Working Directory:	$workspace_loc:/"project name"/src-gen
Arugments:	qdspProgrammer.py -p “serial port” -b “baud rate” ”project name”.bin

3.3.2 Programing interface

The programming interface was devised so it allowed for the programming of individual parts of the program and heap memory, as well as validation of the functionality of the device before programming in started. The interface uses four registers in on either μTosnet of Unity-Link, two to write data and two to receive. The complete command list can be seen in table 3.3.1.

Name	**Transmit**			**Receive**		
	Control		**Data-Out**	**Status**		**Data-In**
start loader	0x81	N/A	N/A	0x81	N/A	N/A
stop loader	0x82	N/A	N/A	N/A	N/A	N/A
write program	0x83	Address	program data	0x83	Address	program data
read program	0x84	Address	N/A	0x84	Address	program data
write heap	0x85	Address	heap data	0x85	Address	heap data
read heap	0x86	Address	N/A	0x86	Address	heap data
validate device	0x87	Feature key	Validation key	0x87	Feature result	Validation result

Table 3.3.1: List of commands used in the program loader.

The control and status registers uses the structure defined in table 3.3.2, that allows extra data to be transmitted along side the command, and the Data-in and -out allows for the transmission of data to the DSP. The Start and Stop loaders commands are used to inform the DSP to go into the programming state. when the programming state is entered the is returned in the status register, but as the loader is stopped when a stop loader command is sent, not change on the status register is possible.

[4]http://pythonhosted.org/pyserial/

Start	**Size[Byte]**	**Description**
0	2	command
2	6	data

Table 3.3.2: Structure of control and Status registers.

The read program and read heap commands uses the address provided in the control data area to return the address and data of a given memory area. When writing a new program to the DSP first the validation must be performed, until that is done the loader is in a read only state. This is done by transmitting a validation command along with a feature key, this feature key is a on-hot encoding of all the instructions required by the code, and a validation key, which structure can be seen in table 3.3.3. These keys are validated on the DSP and the results are returned. If the validation was successful the read-only flag is removed allowing for the programming of the DSP.

Start	**Size[Bit]**	**Description**
0	10	Heap size
10	22	Program size

Table 3.3.3: Structure of the validation key and result.

The write program and heap commands requires the address that is to be written along side the data, when that is received the data is stored in the memory and the new address and data value is returned, allowing for validation of the written data.

Optimization 4

Integrated in the QDSP DSL is a optimizer for optimizing the micro code generated for the QDSP. The optimizer tries to consolidate the use of memory read instructions by setting the memory address of the read instruction in an unused previous memory read, as memory reads requires takes more than one clock cycle. It also tries to remove unnecessary loads of data from the memory to the accumulator, which occurs when a value from f.x. a calculation is stored to a variable and used in the next calculation as well. lastly the optimizer tries to remove unnecessary writes and reservations of heap elements, this is a twofold process one that is run during the incremental processing of the instructions, which removes unnecessary system variables by determining if they are used before being overwritten by another instruction. And a process that runs after all other optimization has completed to ensure that no data is placed on the heap if it is never read.

The optimizer is enabled by default, but can be disabled in the configuration. An example of a source code and the generated output both with and without optimization can be found beneath.

Example 4.0.1: Source

```
config{
  ProjectTitle = firstQDSP
  EnableOptimization = false
}

define in  IO[1]
define out SH[1]
var a
var b
var c
var d

a = in
b = a
c = b
d = c
out = d
```

Example 4.0.2: Without Optimization

```
CIA & A(1),     -- Setting address to 1
NOP & A(0),     -- Do Nothing
CIA & A(0),     -- Copy IO memory address 1 to accumulator, address set in line [0]
CAH & A(0),     -- Copy accumulator to a (Heap0)
CHA & A(0),     -- Copy a (Heap0) to accumulator
CAH & A(1),     -- Copy accumulator to b (Heap1)
CHA & A(1),     -- Copy b (Heap1) to accumulator
CAH & A(2),     -- Copy accumulator to c (Heap2)
CHA & A(2),     -- Copy c (Heap2) to accumulator
CAH & A(3),     -- Copy accumulator to d (Heap3)
CHA & A(3),     -- Copy d (Heap3) to accumulator
CAS & A(1),     -- Copy accumulator to SH memory address 1
RST & A(0),     -- Resets program counter to 0
```

Example 4.0.3: With Optimization

```
CIA & A(1),     -- Setting address to 1
NOP & A(0),     -- Do Nothing
CIA & A(0),     -- Copy IO memory address 1 to accumulator, address set in line [0]
CAS & A(1),     -- Copy accumulator to SH memory address 1
RST & A(0),     -- Resets program counter to 0
```

5 Revision Table

Date	Version	Editor	Description of Revision
23/01/2014	1.0	ankra06 jhenn06	Initial Revision