Document #	Date	Status
LCA-XXXXX	2017-06-18	DRAFT 2

Author(s)

Laurent Le Guillou, Claire Juramy, Eric Aubourg, Stefano Russo, Eduardo Sepulveda, Pierre Antilogus

Subsystem/Office

Systems Integration, Camera Control, Electronics

Document Title

LSST REB Sequencer Language – *User Manual*

Change History Log		
0.1	2017-04-18	First draft
0.2	2017-06-18	Second draft

Contents

1

Introduction

1.1 Purpose and scope of this document

The aim of this document is the description of the REB sequencer programming language, which is the high level language used to define clocking sequences and sequencer programs for the REB FPGA.

This programming language has been developped to facilitate the definition of the REB clocking sequences for all the CCD of the LSST camera, based on the features provided by the REB FPGA (see [7] for a complete description of the REB FPGA programming interface). As the REB FPGA offers a highly flexible programming interface, the dedicated programming language described hereby aims to offer a *human readable* version of this interface, while retaining all of its possibilities.

The sequencer language allows to define simple CCD clocking sequences to clear the CCD / take a single full frame / take a window / and so on; as it is very flexible, it also allows to define more exotic sequences for calibrations and sensor testing, and is heavily used on the LPNHE sensor testing facility, particularly for the optimization of the CCD clocking, and for various sensor tests.

The LSST Camera Control System (CCS) includes the needed tools to compile sequencer programs written for LSST REB sequencer language, and to load them into the REB FPGA memory.

The LSST REB sequencer language is described in details in section 2; to complete this description, several examples are provided in section 3. In section 4 the compilation process and the resulting compiled sequencer file format – ready to be loaded into the REB FPGA memory – are discussed. For reference, section 5 provides the formal grammar of the language in the Backus–Naur form ("BNF").

1.2 Applicable Documents and Reference Documents

The following documents are applicable and form a part of this design document:

Ref #	Document Number	Document Title
[1]	LCA-277	LSST Camera Conceptual Design Report
[2]	LCA-10055	REB/DREB Specification and Design
[3]	LCA-335	DAQ-SRT ICD
[4]	LCA-336	DAQ-CCS ICD
[5]	LCA-50	CCS Specification
[6]	LCA-XXXXX	ASPIC III Specification, Design & User Manual
[7]	LCA-XXXXX	The LSST REB 5 firmware: User Manual
[..]	TBD	TBD

1.3 Acronyms

Acronym	Definition
BNL	Brookhaven National Laboratory
CCD	Charge-Coupled Device
CCS	LSST Camera Control System
FPGA	Field Programmable Gate Array
ICD	Interface Control Document
LPNHE	Laboratoire de Physique Nucleaire et des Hautes Energies
REB	Raft Electronics Board
SLAC	SLAC National Accelerator Laboratory
SRT	Science Raft Tower
TBD	To Be determined
TBR	To Be Reviewed

2

Language description

The LSST REB sequencer language has been designed to offer an easy access to all the sequencer programming features provided by the REB FPGA firmware (see [7]), in order to easily conceive and optimize the clocking sequences for the LSST CCD standard read-out modes, but also for very specific clocking sequences needed for sensor testing activities in the framework of the LSST project.

The elementary bricks of any sequencer program are short sequences of user-defined combinations of output signals (32 output lines are available) and duration of these output configurations, called "*sequencer functions*" or simply "*functions*" hereafter. Before doing any sequencer programming, the user should define these elementary sequencer blocks: the way to do it is described below in section 2.5.

More complex sequencer programs are then built by calling these elementary functions successively, as many times as needed, in the user specified order. The user may structure its sequences by defining several main sequencer programs and auxiliary subroutines, and, in each main program/subroutine, by combining direct calls to elementary functions (**CALL** instruction), and calls to user-defined subroutines (**JSR** "jump to subroutine" instruction). The way to write sequencer programs is described in details in section 2.6.

The syntax of the sequencer language is quite similar to an assembler language, where each elementary instruction is translated into one microprocessor elementary instruction opcode. The FPGA instruction set is very limited, with only four instructions: **CALL**, **JSR**, **RTS** and **END**, and 4 addressing modes (see below in section 2.6.1). The sequencer language offers a few extra instructions and advanced features akin to a preprocessor language, described in section 2.6.6.

2.1 Sequencer file structure

A sequencer program file is divided in several *sections*, each section indicated by a section marker between brackets, and ending with an empty line. The section order is mandatory, but some sections are optional. Here is the typical structure of a sequencer program file:

```
# A comment (everything after a '#' is ignored)
[includes]   # (optional) to includes definitions from other files
  [...]

[constants]  # global parameters, will be substituted at compilation
  [...]

[clocks]  # definition (naming) of the clock channels
  [...]

```

```
[pointers]   # (optional) indirect addressing of functions,
             # subroutines or repetition number
             # (for subroutines or functions)
  [...]

[functions] # elementary sequencer functions definition
  [...]

[subroutines] # (optional) subroutine definitions
  [...]

[mains] # main programs definitions
  [...]

[triggers] # (Acq v2.0 only) 'mains' which could be synchronously
  [...]    # triggered by the LSST Camera Acquisition system.
```

2.2 Including definitions from other files: `[includes]`

The first section, "`[includes]`" is optional; it allows to give a list of other sequencer program files to be included at compilation time. Files will be read and analysed in the provided order, before analysing the current file: any definition may be overwritten and superseded by a later one. The user has to be careful as duplicated definition may be overwritten.

```
[includes]    # optional: to includes definitions from other files
  ../seqfiles/global_constants.seq
  ../seqfiles/CCD_E2V_constants.seq
```

2.3 Sequencer global parameters: `[constants]`

In order to group the specification of the sequencer parameters in one place, and to improve the sequencer programs readability, it is possible (and advised) to define global parameters at the beginning of the sequencer file, in the "`[constants]`" section. Constant values may be positive integers, simple arithmetic expressions combining constants (already defined before) and integers, and durations expressed either in seconds ("`s`"), milliseconds ("`ms`"), microseconds ("`us`") or nanoseconds ("`ns`"). In this later case, durations values will be converted at compilation time in FPGA clock cycles, using the special constant "clockperiod" to do the conversion. If not specified, the "clockperiod" default is 10 ns.

The global parameters definition should follow the syntax shown in the example below:

```
[constants]   # will be substituted in the code at compilation time
  DetectorCols:   576      # Total number of columns in a full readout
  DetectorRows: 2048       # Total number of rows in a full readout
  clockperiod:     10 ns   # FPGA clock period (required)
  TimeP:         5000 ns   # Base time element of parallel transfers
  BufferP:       2500 ns   # Parallel transfer buffer time
  RampTime:       320 ns   # ASPIC ramp time
  SegCols:        512      # Number of columns of the sensors
  SerCols:        522      # Size of serial register
  PreScan:   SerCols - SegCols   # Prescan pixels
```

2.4 Sequencer output lines: [clocks]

To simplify the sequencer programming, the clock output lines (or clock channels) may me named, and the naming scheme is described in the "**clocks**" section of the sequencer file:

```
[clocks]   # clock channels (output lines)
    P1:    8  # Parallel clock 1
    P2:    9  # Parallel clock 2
    P3:   10  # Parallel clock 3
    P4:   11  # Parallel clock 4
    S1:    4  # Serial clock 1
    S2:    5  # Serial clock 2
    S3:    6  # Serial clock 3
    RG:    7  # Serial reset clock
    CL:    3  # ASPIC clamp
    RST:   2  # ASPIC integrator reset
    RD:    1  # ASPIC ramp-down integration
    RU:    0  # ASPIC ramp-up integration
    TRG:  12  # ADC sampling trigger
    SOI:  13  # Start of image
    EOI:  14  # End of image
    SHU:  16  # Shutter TTL (for testing only)
```

This naming scheme should of course be adapted for each CCD type (E2V/ITL).

In the example above, output line 16 ("SHU") is connected to a shutter controller, to command the shutter on sensor testing facilities. This is only for testbench tests, The LSST camera shutter will not be controlled through the REB boards.

2.5 The elementary clocking sequences: [functions]

The sequencer is divided in 16 user defined functions: each function generates a synchronous sequence of output signals. A sequencer program defines the execution order of the loaded functions.

The function is the basic element of the sequencer: it is defined by the user to generate a time sequence of 32 output signals. Each function is divided into up to 16 *time slices* : each time slice is defined by its duration and the states of the outputs (0 = down / 1 = up). Output transitions happen between two successives time slices.

The time slice duration may be specified explicitely or by using a global parameter (see section... constants above). The duration may be specified as a number of FPGA clock cycles, or in seconds (`'s'`), milliseconds ('ms'), microseconds ('us') or nanoseconds ('ns'); in the later case, the duration will be converted in FPGA clock cycles using the clockperiod parameter (default is 10 ns). The time slice duration may be specified with a global parameter defined before (see above in 2.3).

If some outputs should stay in the same defined state (0/1) during the whole sequencer function, this may be specified with the **constants** keyword.

```
[functions]
  <function_name>     #   comment
    clocks:        <list of affected output lines>
    slices:
      <duration>  =  0/1, 0/1, 0/1, ...
      <duration>  =  0/1, 0/1, 0/1, ...
```

```
      <duration>  =  0/1, 0/1, 0/1, ...
      [...]
    constants:    <line>=1, <line>=0, <line>=1,...
```

In the following example, two functions are defined: the "`Default`" one (which should always be the first one defined), and the "TransferLine" function which changes the parallel lines in order to move the CCD charges one line down (for the 4-phase E2V CCD).

```
[functions]
  Default:  # Default state when not operating
    clocks:         P2, P3, S1, S2, RG, CL, RST
    slices:
      1 us        =  1,  1,  1,  1,  1,  1,  1

  TransferLine:  # Single line transfer
    clocks:         P1, P2, P3, P4, RG, CL
    slices:
      BufferP     =  0,  1,  1,  0,  1,  0
      TimeP       =  0,  0,  1,  1,  1,  0
      TimeP       =  1,  0,  0,  1,  1,  0
      TimeP       =  1,  1,  0,  0,  1,  0
      BufferP     =  0,  1,  1,  0,  0,  0
      7540 ns     =  0,  1,  1,  0,  0,  0
    constants:  S1=1, S2=1, RST=1
```

Functions will be automatically numbered in the order they are defined in the sequencer file. The "`Default`" (function #0) one should always be the first defined.

The first time slice of the first function (#0, here named "`Default`") has a particular role: it describe the default state of the REB outputs when the sequencer is not in operating mode. The outputs will fall back to this default state when any sequence execution ends.

2.6 Writing the sequencer program: `[mains]` and `[subroutines]`

The REB FPGA provides a program memory of 1024 elementary instructions, allowing the user to store several sequencer programs to be executed. Each sequence is defined into a sequencer program: to simplify the programming, a given program may call subroutines, which may in turn call other subroutines as well, and so on.

The top level sequencer programs, also called "mains", are defined into the "[mains]" section, while auxiliary subroutines are grouped into the "[subroutines]" section. The syntax for both is the same: the only difference is that a "main" program should end with the final "END" instruction, while an auxiliary subroutine should end with the "RTS" (ReTurn from Subroutine) instruction.

Only the top level "mains" may be triggered by the acquisition system (see also TRIGGER and section 2.7).

2.6.1 Program instructions

The REB FPGA offers 4 different elementary instructions:

- "**`CALL`**" to call an elementary sequencer function (one of the 16 sequencer function, see sec. func above). Only "**`CALL`**" instructions really modify the output signals. An optional "**`repeat(...)`**"

argument allows to specify how many times this call should be performed (default is once). There are several way to perform a "**CALL**", described below in section 2.6.2, and "**CALL**" instructions are translated into opcodes `0x1` to `0x4` depending on the addressing mode (see below and [7], section 8.4).

- "**JSR**" to jump to an auxiliary subroutine. The specified subroutine will be executed, and the program will resume on the next instruction. "**JSR**" jumps have no action by themselves on the output signals. As for the "**CALL**" instruction, an optional "**repeat(...)**" argument allows to specify how many times the invoked subroutine should be executed (default is once). There are several way to invoke a "**JSR**", and "**JSR**" instructions are translated into opcodes `0x5` to `0x8` depending on the addressing mode, see below in section 2.6.3, and [7], section 8.4.
- "**RTS**" to end an auxiliary subroutine and go back where the subroutine has been called. This instruction is translated in opcode `0xE` at compilation time (see [7], section 8.4).
- "**END**" to end the sequencer program. This instruction is translated as opcode `0xF` at compilation time (see [7], section 8.4).

2.6.2 Calling a sequencer function: direct and indirect addressing

To provide flexibility, and to avoid writing several alternative programs to perform very similar tasks, the REB FPGA offers different addressing modes to perform a "**CALL**" to a sequencer function.

A sequencer function may be called directly, by providing its `#function_id` (in [0-15]). It may also be called directly by using instead its symbolic name (for instance, `ReadPixel`), which will be substituted at compilation time. Each function call may be repeated by specifying the repetition number with the keyword **repeat**, as below:

```
CALL   3                 # call func(3), once
CALL   3    repeat(10)   # call func(3), 10 times

CALL   TransferLine      # call func TransferLine, once
CALL   TransferLine    repeat(10) # idem, 10 times
CALL   TransferLine    repeat(DetectorRows) # global param.
```

A sequencer function may also be called through an indirect addressing mode, by using a "*function pointer*" (**REP_FUNC**) which stores the function #id to be called. The pointer has to be defined before in the pointers section (see section 2.6.5), and may be modified later by the user. In that case, the indirect subroutine adressing should be indicated by prefixing an "**@**" to the pointer name. For instance,

```
[pointers]
  [...]
  # Function to use during exposure: SerialFlush or ExposureNoFlush
  PTR_FUNC   Exposure   ExposureNoFlush

[subroutines]
  [...]
  Exposure25ms:   # Repeat exposure function for 25 ms
    CALL     @Exposure    repeat(13441)
    RTS
```

In this example, the pointer value of the function pointer `Exposure` may be modified on the fly between 2 sequence executions, by writing its new value at its address in the FPGA memory, without the need to reload the whole sequencer.

In the same way, the repetition number (argument of the **repeat** keyword, which define how many times a given function should be called in a **CALL** instruction) may also be defined through a repetition pointer (**REP_FUNC**), which can also be modified in the FPGA memory without reloading the sequencer. The same syntax (with a "**@**") is used:

```
[pointers]
  [...]
  # Function to use during exposure: SerialFlush or ExposureNoFlush
  PTR_FUNC   Exposure   ExposureNoFlush
  [...]
  # Repetitions of SerialFlush function during FlushRegister
  REP_FUNC   FlushTime 100000

[subroutines]
  [...]
  Exposure25ms:  # Repeat exposure function for 25 ms
    CALL     @Exposure     repeat(13441)
    RTS

  FlushRegister:   # Flushing serial register
    CALL     SerialFlush   repeat(@FlushTime)
    RTS
```

Of course, both (function and repetition) may be specified by indirect addressing, as in the following example:

```
[pointers]
  [...]
  # Function to use during exposure: SerialFlush or ExposureNoFlush
  PTR_FUNC   Exposure   ExposureNoFlush
  [...]
  # Number of repetition to get a 25ms delay
  REP_FUNC   ExpNcycles 13441

[subroutines]
  [...]
  Exposure25ms:  # Repeat exposure function for 25 ms
    CALL     @Exposure     repeat(ExpNcycles)
    RTS
```

The 4 modes of addressing for the "**CALL**" instruction are translated into 4 different opcodes values at compilation time (see [7], section 8.4):

Keyword	Function addressing	Repeat addressing	Opcode
CALL	direct	direct	0x01
CALL / **CALLP**	indirect	direct	0x02
CALL / **CALLREP**	direct	indirect	0x03
CALL / **CALLPREP**	indirect	indirect	0x04

For function calls, the number of repetitions may be specified as infinite (infinite loop), using the special word "`infinity`":

```
[mains]
  InfiniteWait:  # Slow flushing on infinite loop
    CALL     SlowFlush    repeat(infinity)
    END
```

Such a sequencer program can only be stopped at execution time using special **STEP** or **STOP** triggers.

2.6.3 Jumping to a subroutine: direct and indirect addressing

As for the "**CALL**" instruction, the REB FPGA offers different addressing modes to perform a jump to a subroutine with the "**JSR**" instruction.

A subroutine may be called directly, by providing the address of the subroutine first instruction in the program memory, relatively to the beginning of the program memory space. A subroutine may also be called directly by using instead its symbolic name (for instance, `ReadPixel`), which will be substituted by the subroutine address at compilation time.

As for jumps, each subroutine call may be repeated by specifying the repetition number with the keyword **repeat**. Contrary to functions calls, a subroutine cannot be repeated infinitely.

```
  JSR    0x080       # jump to subroutine stored
                     # at address 0x080, once

  JSR    ReadLine    # jump to subroutine ReadLine, once

  JSR    ReadLine    repeat(10)   # jump to ReadLine, 10 times
  JSR    ReadLine    repeat(DetectorLines) # global param.
```

A subroutine may also be called through an indirect addressing mode, by using a "*subroutine pointer*" (**REP_SUBR**) which stores the memory address of the subroutine to be called. The pointer has to be defined before in the pointers section (see section 2.6.5), and may be modified later by the user. The syntax is similar to **CALL** instructions: the indirect addressing is specified by prefixing pointers by a "**@**" symbol. For instance,

```
[pointers]
  [...]
  PTR_SUBR  MyClearCCD    0x0f0  # subroutine address, or
  PTR_SUBR  MyClearCCD    FastClear # subroutine name

[subroutines]
  [...]
  Clear:   # Clearing only
    JSR      @MyClearCCD   repeat(10)
    END
```

In the same way, the repetition number (argument of the **repeat** keyword, which define how many times the subroutine should be invoked may also be defined through a repetition pointer (**REP_SUBR**), which can also be modified directly in the FPGA memory:

```
[pointers]
  # Number of rows to skip before window
  REP_SUBR  PreRows       0
  # Number of rows of the window
```

```
  REP_SUBR  ReadRows  2048
  # Number of rows after window
  REP_SUBR  PostRows     0
  [...]

[subroutines]
  [...]
  ReadFrame:  # Readout and acquisition of a CCD frame (window)
    JSR     FlushLine         repeat(@PreRows)
    JSR     FlushRegister
    CALL    StartOfImage
    JSR     WindowLine        repeat(@ReadRows)
    CALL    EndOfImage
    JSR     FlushLine         repeat(@PostRows)
    RTS
```

Of course, both indirect addressing modes may be used for the same **JSR** instruction:

```
[pointers]
  # Number of rows of the window
  REP_SUBR  ReadRows      2048
  # Pointer to the subroutine to read a line
  PTR_SUBR  MyWindowLine  SpecialLineReadout
  [...]

[subroutines]
  [...]
  ReadFrame:  # Readout and acquisition of a CCD frame (window)
    JSR     FlushLine         repeat(@PreRows)
    JSR     FlushRegister
    CALL    StartOfImage
    JSR     @MyWindowLine     repeat(@ReadRows) # both indirect
    CALL    EndOfImage
    JSR     FlushLine         repeat(@PostRows)
    RTS
```

The 4 modes of addressing for the "**JSR**" jump-to-subroutine instruction are translated into 4 different opcodes values at compilation time (see [7], section 8.4):

Keyword	Function addressing	Repeat addressing	Opcode
JSR	direct	direct	0x05
JSR / **JSP**	indirect	direct	0x06
JSR / **JSREP**	direct	indirect	0x07
JSR / **JSPREP**	indirect	indirect	0x08

At the implementation level in the FPGA microcode, there is a limitation on the number of nested subroutine calls: at maximum, 15 nested subroutine calls may be performed. The user has to be careful to avoid reaching this limit, as the resulting behavior may be unpredictable otherwise.

2.6.4 Defining a main program / auxiliary subroutine

Main programs ("mains") and subroutines are defined as blocks of instructions (one instruction per line), with the following syntax:

```
[subroutines]

  <subroutine_name>:  # comment
    <instruction>
    <instruction>
    [...]
    RTS

  [...]

[mains]

  <main_name>:  # comment
    <instruction>
    <instruction>
    [...]
    END

  [...]
```

The main programs should be defined in the "**[mains]**" section and should end with the **END** instruction, while the auxiliary subroutines should be defined in the "**[subroutines]**" section and end with a **RTS** instruction. Otherwise, the syntax for both is the same. Here is an example for a subroutine to read a CCD frame:

```
[subroutines]
  ReadFrame:  # Readout of a CCD frame (window)
    JSR     FlushLine        repeat(@PreRows)   # PreRows is a rep. ptr
    JSR     FlushRegister
    CALL    StartOfImage
    JSR     WindowLine       repeat(@ReadRows)
    CALL    EndOfImage
    JSR     FlushLine        repeat(@PostRows)
    RTS
```

2.6.5 Indirect addressing: pointers

To allow indirect addressing modes for **CALL** and **JSR** instructions, 4 types of pointers are available: function pointers (**PTR_FUNC**, pointing to a function id), subroutine pointers (**PTR_SUBR**, pointing to a subroutine address), function repetition pointers (**REP_FUNC**, pointing to a repetition number for a function), and subroutine repetition pointers (**REP_SUBR**, pointing to a repetition number for a subroutine).

Before using a given pointer, it has to be declared and defined in the dedicated "**[pointers]**" section, following this syntax:

```
[pointers]
  # Number of columns to read
  REP_FUNC   ReadCols       576
  # Number of rows of the window
  REP_SUBR   ReadRows      2048
  # Number of full CCD clears before acquiring
```

```
REP_SUBR  CleaningNumber  2
# Subroutine to use for clearing the frame
PTR_SUBR  CleaningSubr  MixedFlushLine
# Function to use during shutter closing:
# SerialFlush or DarkNoFlush
PTR_FUNC    ClosingFunc    DarkNoFlush
[...]
```

For each pointer type, up to 16 pointers may be defined.

2.6.6 Advanced sequencer programming

The sequencer language offers a few "advanced features", designed for very specific uses during sensor testing: simple mathematical (arithmetic) expressions can be used, and three extra instructions: **SET**, **IF** and **WHILE** are provided to simplify the writing of some complex sequencer programs.

All these features are similar to pre-processing features offered by other programming languages (C/C++ preprocessor for instance): mathematical expressions, as well as **SET**, **IF** and **WHILE** statements are processed at compilation time, and are not at all executed by the FPGA at runtime (the expression evaluation, and the **SET**, **IF** and **WHILE** instructions do not exist in the REB FPGA programming interface).

The user should keep this in mind when writing complex sequencer programs using these language features.

Mathematical expressions

When defining the value of a global parameter (section 2.3), a local parameter (see below), or a repetition number, it is possible to provide a simple arithmetic expression instead of an integer number. Allowed expressions may combine integers, global or local parameters, the mathematical operators "**+**" (addition), "**-**" (subtraction) and "*****" (multiplication). Comparison operators ("**==**", "**!=**", "**<**","**<=**","**>**","**>=**") may also be used, but only at top level.

These mathematical expressions are evaluated at compilation time, and should not include indirect adressing values ("pointers", see 2.6.5).

Local parameters

With the "**SET**" instruction, the sequencer language offers the ability to define local parameters: these parameters will only be valid in the subroutine/main block where they have been defined. During the compilation process, these parameters will be substituted by their current value.

The syntax is the following:

```
SET  <localparameter>   <expr>
```

The local parameter value may be any valid expression (see above):

```
[subroutines]
  [...]
  MySpecialReadout:
    SET  windowsize   325
    SET  MyCols       4 * (2 + DetectorCols) - 1
    SET  MyParam      MyCols - 3 * DetectorCols
```

```
  [...]
  CALL ReadPixel    repeat(MyParam)
  [...]
```

Local parameters may be used in the same way than global parameters, but only in the subroutine/-main block where they have been defined.

Conditional blocks

The sequencer language offers a very simple conditional block with the "**IF**" instruction, which is analog to `#IFDEF` statements in the C/C++ preprocessor language. The "**IF**" syntax is the following:

```
IF <expr> THEN
  <instruction>
  <instruction>
  [...]
FI
```

The expression is evaluated at compilation time, and if the value is not zero, the instructions between the IF and FI statements will be compiled; otherwise they are ignored. Here is an example to illustrate the "**IF**" syntax:

```
[...]
SET  PostCols     60
[...]
IF PreCols+Cols+PostCols < DetectorCols THEN
  CALL ReadPixel  repeat(Cols)
  IF PostCols > 10 THEN
    CALL ReadPixel  repeat(PostCols)
  FI
FI
```

One possible use of this feature is to define global parameters specifying the CCD model (E2V or ITL), and to compile specific parts of the sequencer code depending of the target CCD model;

```
[constants]
  [...]
  CCD_E2V     1   #  1(True) if the CCD is an E2V one
  CCD_ITL     0   #  1(True) if the CCD is an ITL one
  [...]
```

And in any subroutine/main program, some code portions may be compiled only for one CCD model:

```
[...]
IF CCD_E2V THEN
  CALL FastFlush  repeat(10)
FI
[...]
```

Conditional loops

The sequencer language offers a very simple conditional loop with the "**WHILE**" instruction. Its syntax is shown below:

```
WHILE <expr> DO
  <instruction>
  <instruction>
  [...]
DONE
```

As long as the specified expression <expr> is evaluated as non-zero, the block of instructions between **WHILE** and **DONE** is repeated. Loops are unrolled at compilation time. For instance, the following code fragment:

```
SET   MaxLines   6
SET   iLine      1
WHILE iLine < MaxLines DO
  CALL   ReadLine   repeat(iLine)
  CALL   SerialFlush
  SET    iLine   iLine + 1
DONE
```

is equivalent to:

```
CALL    ReadLine   repeat(1)
CALL    SerialFlush
CALL    ReadLine   repeat(2)
CALL    SerialFlush
CALL    ReadLine   repeat(3)
CALL    SerialFlush
CALL    ReadLine   repeat(4)
CALL    SerialFlush
CALL    ReadLine   repeat(5)
CALL    SerialFlush
```

Of course, **IF** and **WHILE** blocks may be mixed and nested.

To avoid compilation failure in the case of an (accidental) infinite loop, there is a limit of 1000 loop iterations for any **WHILE** loop. When this limit is reached, the compilation process stops and fails. Anyway, as the program memory is limited to 1024 elementary instructions, this limit should never be reached if the user aims to write a program which could fit in the FPGA program memory.

2.7 Sequencer trigger mechanism

2.7.1 Main program pointer

The trigger mechanism described hereby is only working with LSST Camera Acquisition version 1.0. This is no longer valid in Acquisition version 2.0: see section 2.7.2.

As the sequencer language allows to define several main programs, there should be a way to specify which one should be started when the FPGA receives the trigger signal (which is done by setting the bit 2 in the TRIGGER register, at address `0x8`; see [7] section 7.6).

To select the program to be run, a special pointer named "Main" could be defined in the "**[pointers]**" section, to specify which "main" program should be launched when the sequencer is triggered:

```
[pointers]
  REP_FUNC     PreCols       50
  REP_FUNC     ReadCols     256
  [...]
  MAIN         Main         Bias  # Default main program is Bias
```

In the example above, the address of the first instruction of the Bias main program will be stored into the special Main pointer, which is stored at address `0x340000` (see [7] section 8.4.13). If no Main pointer has been defined in the sequencer file, the first defined main program will be selected as the default one.

Once the sequencer has been loaded into the FPGA program memory, the main pointer may be modified by the user by writing the address of another main program into the Main pointer. That way, the user may switch to another defined main program. This mechanism allows the user to run many different sequencer programs, as long as they are loaded into the FPGA program memory.

2.7.2 Triggers: `[triggers]`

WARNING: this section is still under discussion.

The trigger mechanism described here replaces the previous one for the LSST Camera Acquisition version 2.0.

As the user may define several main sequencer program in the "`[mains]`" section, there should be a way to select which one should be triggered.

With the LSST Camera Acquisition version 2.0, the acquisition system may trigger 8 different actions, numbered from 0 to 7, by sending a synchronous signal to a subset of REB boards. It is up to the sequencer user to define which main program should be run for each of this signals. This is done in the "**`[triggers]`**" section at the end of the sequencer program file:

```
[triggers]
  0: Clear
  1: Bias
  2: Dark
  3: Acquisition
  4: VariantAcquisition
  5: AnotherMain
  # 6: STEP  (RESERVED)
  # 7: STOP  (RESERVED)
```

Trigger signals 0 to 5 are available and could be associated to any main program defined in the sequencer program file. Trigger signals 6 and 7 are reserved and are associated with the STEP and STOP (see [7] section 8.4.5.2).

The trigger address table is stored at addresses 0x340000–0x340005 (TBR), and the user may modify this table without reloading the whole sequencer into the FPGA memory.

3

Sequencer examples

3.1 Simple sequencer

As an example, we present below a simple sequencer for the E2V, offering bias frames (`Bias`), normal frames (`Acquisition`), with several flushing modes for the serial register. Window frame could be taken by adapting the values of the `PreCols`, `ReadCols`, `PostCols` and `PreRows`, `ReadRows`, `PostRows` repetition pointers, and this can be done for each individual frame without reload the whole sequencer.

In this setup, it is assumed that the shutter opening and closing is controlled by the REB itself, by sending a signal on line 16 (SHU line), as on the LPNHE/Paris LSST testbench.

```
# REB3 timing for E2V CCD, in new REB sequencer format
# new baseline sequencer with overlap in parallel clocks
# 20170119, C. Juramy.

[constants]  # will be substituted in the code at compilation time, if used
    SegRows:        2002    # Number of rows of the sensor
    SegCols:         512    # Number of columns of the sensors
    SerCols:         522    # Size of serial register
    DetectorCols:    576    # Total number of columns in a full readout
    DetectorRows:   2048    # Total number of rows in a full readout
    TimeP:             5000 ns    # Base time element of parallel transfers
    OverlapP:          1000 ns    # Overlap at three phases in parallel transfer
    BufferP:           2500 ns    # Parallel transfer buffer time
    TimeS:             300 ns     # Base element of serial transfers
    BufferS:            80 ns     # Buffer for serial clock crossing
    RampTime:          320 ns     # ASPIC ramp time
    ISO1:              130 ns     # Time between end of ASPIC clamp/reset and start of RD
    ISO2:              320 ns     # Time between S3 down and start of ASPIC RU
    FlushS:            540 ns     # Base element for flushing the serial register
    clockperiod:       10 ns      # FPGA clock period (required by the interpreter)
    ElemExposure:      25 ms      # Duration of the elementary exposure subroutine

[clocks]  # clock channels
    P1:    8  # Parallel clock 1
    P2:    9  # Parallel clock 2
    P3:   10  # Parallel clock 3
    P4:   11  # Parallel clock 4
    S1:    4  # Serial clock 1
    S2:    5  # Serial clock 2
    S3:    6  # Serial clock 3
    RG:    7  # Serial reset clock
    CL:    3  # ASPIC clamp
    RST:   2  # ASPIC integrator reset
    RD:    1  # ASPIC ramp-down integration
    RU:    0  # ASPIC ramp-up integration
    TRG: 12  # ADC sampling trigger
    SOI: 13  # Start of image
    EOI: 14  # End of image
    SHU: 16  # Shutter TTL (for testing only)

[pointers]  # can define a pointer to a function or to a repetition number
```

```
        # (for subroutines or functions)
REP_FUNC    PreCols      300   # Number of columns to skip
                               # before readout window, including prescan
REP_FUNC    ReadCols      50   # Number of columns to read
REP_FUNC    PostCols     226   # Number of columns to discard after window
                               # (it is up to the user that total columns = 576)
REP_FUNC    OverCols      50   # Number of columns acquired after line is read
                               #for baseline subtraction
REP_SUBR    ExposureTime  80   # Duration of exposure in units of 25 ms
REP_SUBR    PreRows     1000   # Number of rows to skip before window
REP_SUBR    ReadRows      50   # Number of rows of the window
REP_SUBR    PostRows     970   # Number of rows after window
                               # (it is up to the user that total lines = 2048)
REP_SUBR    CleaningNumber  2   # Number of full CCD clears before acquiring a frame
PTR_SUBR    CleaningSubr      FlushLine   # Subroutine to use for clearing the frame
PTR_FUNC    Exposure    ExposureFlush   # Function to use during exposure:
                                        # SerialFlush or ExposureFlush
                                        # or ExposureNoFlush or DarkNoFlush
                                        # (in addition to the periodic flushing)
PTR_FUNC    ClosingFunc   SerialFlush   # Function to use during shutter closing:
                                        # SerialFlush or DarkNoFlush
REP_FUNC    ShutterTime   50000   # Repetitions of ClosingFunc function
                                  # during ShutterClose (approx 100 ms)
REP_FUNC    FlushTime     50000   # Repetitions of SerialFlush function
                                  # during FlushRegister
REP_SUBR    FlushLines      100   # Repetitions of the fake readout lines
                                  # during FlushRegister

[functions]
    Default:  # Default state when not operating
      clocks:          P2, P3, S1, S2, RG, CL, RST
      slices:
         1 us        =  1,  1,  1,  1,  1,  1,  1

    TransferLine:  # Single line transfer
      clocks:          P1, P2, P3, P4
      slices:
         BufferP     = 0,  1,  1,  0
         OverlapP    = 0,  1,  1,  1
         TimeP       = 0,  0,  1,  1
         OverlapP    = 1,  0,  1,  1
         TimeP       = 1,  0,  0,  1
         OverlapP    = 1,  1,  0,  1
         TimeP       = 1,  1,  0,  0
         OverlapP    = 1,  1,  1,  0
         1000 ns     = 0,  1,  1,  0
         7540 ns     = 0,  1,  1,  0  # made it longer to match e2v timing
      constants:     S1=1, S2=1

    ParallelFlush: # Single line transfer with all serial register clocks high to flush it
      clocks:          P1, P2, P3, P4
      slices:
         7500 ns     = 0,  1,  1,  0
         OverlapP    = 0,  1,  1,  1
         15000 ns    = 0,  0,  1,  1
         OverlapP    = 1,  0,  1,  1
         15000 ns    = 1,  0,  0,  1
         OverlapP    = 1,  1,  0,  1
         15000 ns    = 1,  1,  0,  0
         OverlapP    = 1,  1,  1,  0
         36000 ns    = 0,  1,  1,  0  # made it longer to match e2v timing
         7500 ns     = 0,  1,  1,  0
      constants: S1=1, S2=1, S3=1, RG=1, RST=1

    ReadPixel:  # Single pixel read
      clocks:          RG, S1, S2, S3, CL, RST, RD, RU, TRG
      slices:
          50 ns      = 1,  0,  1,  0,  0,  0,  0,  0,  1
         150 ns      = 1,  0,  1,  0,  0,  0,  0,  0,  0
         BufferS     = 1,  0,  1,  1,  0,  1,  0,  0,  0
         BufferS     = 0,  0,  0,  1,  0,  1,  0,  0,  0
         250 ns      = 0,  0,  0,  1,  1,  1,  0,  0,  0
         ISO1        = 0,  0,  0,  1,  0,  0,  0,  0,  0
         RampTime    = 0,  0,  0,  1,  0,  0,  1,  0,  0
         BufferS     = 0,  1,  0,  1,  0,  0,  0,  0,  0
         ISO2        = 0,  1,  0,  0,  0,  0,  0,  0,  0
```

```
    RampTime     = 0,  1,  0,  0,  0,  0,  0,  1,  0
    BufferS      = 0,  1,  1,  0,  0,  0,  0,  0,  0
  constants:     P2=1, P3=1

StartOfImage:  # Signals start of frame to be recorded
  clocks:          SOI
  slices:
    1600 ns      = 0  # lets ADC finish previous conversion and transfer
    100 ns       = 1
    100 ns       = 0
  constants:     P2=1, P3=1, S1=1, S2=1, RG=1

EndOfImage:  # Signals end of frame to be recorded
  clocks:          EOI
  slices:
    1600 ns      = 0  # lets ADC finish conversion and transfer
    100 ns       = 1
    100 ns       = 0
  constants:     P2=1, P3=1, S1=1, S2=1, RG=1

SerialFlush:  # Single pixel flush with timing set by FlushS parameter
  clocks:          RG, S1, S2, S3
  slices:
     FlushS      = 1,  0,  1,  0
     BufferS     = 1,  0,  1,  1
     FlushS      = 0,  0,  0,  1
     BufferS     = 0,  1,  0,  1
     FlushS      = 0,  1,  0,  0
     BufferS     = 0,  1,  1,  0
  constants:     P2=1, P3=1, RST=1

ExposureFlush:  # Exposure while flushing serial register (testing only)
                # same timing as SerialFlushReg
  clocks:          RG, S1, S2, S3
  slices:
     FlushS      = 1,  0,  1,  0
     BufferS     = 1,  0,  1,  1
     FlushS      = 1,  0,  0,  1
     BufferS     = 1,  1,  0,  1
     FlushS      = 1,  1,  0,  0
     BufferS     = 1,  1,  1,  0
  constants:     P2=1, P3=1, RST=1, SHU=1

DarkNoFlush:  # Dark without flushing serial register
              # same timing as SerialFlushReg
  clocks:          RG, S1, S2, S3
  slices:
     FlushS      = 1,  1,  1,  1
     BufferS     = 1,  1,  1,  1
     FlushS      = 1,  1,  1,  1
     BufferS     = 1,  1,  1,  1
     FlushS      = 1,  1,  1,  1
     BufferS     = 1,  1,  1,  1
  constants:     P2=1, P3=1, RST=1

ExposureNoFlush:  # Exposure without flushing serial register (testing only),
                  # same timing as SerialFlushReg
  clocks:          RG, S1, S2, S3
  slices:
     FlushS      = 1,  1,  1,  1
     BufferS     = 1,  1,  1,  1
     FlushS      = 1,  1,  1,  1
     BufferS     = 1,  1,  1,  1
     FlushS      = 1,  1,  1,  1
     BufferS     = 1,  1,  1,  1
  constants:     P2=1, P3=1, RST=1, SHU=1

SlowFlush:  # Simultaneous serial and parallel flush, slow (waiting pattern)
  clocks:          RG, S1, S2, S3, P1, P2, P3, P4
  slices:
     TimeP       = 1,  0,  1,  0,  0,  1,  1,  0
     TimeP       = 0,  0,  0,  1,  0,  1,  1,  0
     TimeP       = 0,  1,  0,  0,  0,  1,  1,  0
     TimeP       = 0,  0,  1,  0,  0,  1,  1,  0
     TimeP       = 0,  0,  0,  1,  0,  1,  1,  0
     TimeP       = 0,  1,  0,  0,  0,  1,  1,  0
```

```
        TimeP       = 0,  0,  1,  0,  0,  1,  1,  0
        TimeP       = 0,  0,  0,  1,  0,  1,  1,  0
        TimeP       = 0,  1,  0,  0,  0,  1,  1,  0
        TimeP       = 1,  0,  1,  0,  0,  1,  1,  0
        20000 ns    = 1,  0,  1,  0,  0,  0,  1,  1
        20000 ns    = 1,  0,  1,  0,  1,  0,  0,  1
        20000 ns    = 1,  0,  1,  0,  1,  1,  0,  0
        20000 ns    = 0,  0,  1,  0,  0,  1,  1,  0
    constants:     CL=1, RST=1

[subroutines]
#
# Line-level operations ---------------------------------------------
#
# including several options to flush lines

   FlushLine:  # Transfer line with all serial clocks and reset high
       CALL    ParallelFlush
       RTS

   PixelFlushLine:  # Transfer line and flush it pixel by pixel
       CALL    TransferLine
       CALL    SerialFlush      repeat(DetectorCols)
       RTS

   WindowLine:  # Line readout
       CALL    TransferLine
       CALL    SerialFlush      repeat(@PreCols)
       CALL    ReadPixel        repeat(@ReadCols)
       CALL    SerialFlush      repeat(@PostCols)
       RTS

   WindowWithOverscan:  # Line readout adding pixels in the overscan
       CALL    TransferLine
       CALL    SerialFlush      repeat(@PreCols)
       CALL    ReadPixel        repeat(@ReadCols)
       CALL    SerialFlush      repeat(@PostCols)
       CALL    ReadPixel        repeat(@OverCols)
       RTS

#
# Frame-level readout operations -----------------------------------
#

   CloseShutter:  # Gives time for shutter to close
                  # (to be adapted depending on setup)
       CALL    @ClosingFunc     repeat(@ShutterTime)
       RTS

   FlushRegister:  # Flushing serial register from accumulated charges
       CALL    SerialFlush      repeat(@FlushTime)
       RTS

   ReadFrame:  # Readout and acquisition of a CCD frame (window)
       JSR     FlushLine        repeat(@PreRows)
       JSR     FlushRegister
       CALL    StartOfImage
       JSR     WindowLine       repeat(@ReadRows)
       CALL    EndOfImage
       JSR     FlushLine        repeat(@PostRows)
       RTS

   FakeFrame:  # Readout of a CCD frame (window) with no data output
       JSR     FlushLine        repeat(@PreRows)
       JSR     FlushRegister
       JSR     WindowLine       repeat(@ReadRows)
       JSR     FlushLine        repeat(@PostRows)
       RTS

#
# Exposure operations -----------------------------------------------
#

   Exposure25ms:  # Repeat exposure function for 25 ms
       CALL    @Exposure        repeat(13441)
       RTS
```

```
    ClearCCD:  # Clear CCD once
        JSR     @CleaningSubr       repeat(DetectorRows)
        RTS

    AcquireFrame:  # Operations to expose (or not) a CCD frame
        JSR     ClearCCD            repeat(@CleaningNumber)
        JSR     Exposure25ms        repeat(@ExposureTime)
        JSR     CloseShutter
        RTS


[mains]
    RawBias:  # Bias without clearing first
        JSR     ReadFrame
        END

    Clear:  # Clearing only
        JSR     ClearCCD        repeat(@CleaningNumber)
        END

    Bias:  # Bias after clearing up CCD content
        JSR     ClearCCD        repeat(@CleaningNumber)
        JSR     ReadFrame
        END

    Acquisition:  # One acquisition (exposure or dark)
        JSR     AcquireFrame
        JSR     ReadFrame
        END

    NoAcquisition:  # Simulates acquisition without storing image (for debugging)
        JSR     AcquireFrame
        JSR     FakeFrame
        END

    InfiniteWait:  # Slow flushing on infinite loop
        CALL    SlowFlush    repeat(infinity)
        END

    Dark:  # copied from Acquisition, for compatibility with previous sequences
        JSR     AcquireFrame
        JSR     ReadFrame
        END
```

3.2 Special flat sequence

As a second example, we present here a set of sequencer extra subroutines allowing to make a very uniform flat frame by reading the CCD while it is still illuminated. In a first step, all lines are flushed out at the normal readout speed, but as the `StartOfImage` function has not been called, no pixels are sent to the LSST acquisition system. Then, a `StartOfImage` is sent, and we proceed to a normal readout of the CCD frame. In the resulting recorded frame, the flux in each pixel is the average of the CCD illumination along the whole CCD column, resulting in a very uniform frame, even if the illumination pattern is not very uniform. This type of sequence may prove useful for gain measurements, for instance.

```
[...]
[subroutines]
    [...]

    AcquireFrame:  # Operations to expose (or not)
        JSR     ClearCCD            repeat(@CleaningNumber)
        JSR     Exposure25ms        repeat(@ExposureTime)
        JSR     CloseShutter
        RTS
```

```
    FlatFrame:  # Special flat frame: move all lines out,
               # then read a normal frame
        JSR     FlushRegister
        JSR     WindowLine      repeat(@ReadRows)
        CALL    StartOfImage
        JSR     WindowLine      repeat(@ReadRows)
        CALL    EndOfImage
        RTS

[mains]
    [...]

    FlatAcquisition:
        JSR     AcquireFrame
        JSR     FlatFrame
        END
```

3.3 Reverse clocking to measure non-linearity

In this third example, we present here a set of sequencer subroutines which may prove useful to measure the non-linearity at very low illumination levels. The sequence `LinearityAcquisition` runs as follow (with light on, and no shutter):

1. The whole CCD is cleared.
2. Then the following sequence is repeated:
   (a) Using reverse parallel transfer, we move 100 lines up; No pixel is read yet.
   (b) The sequencer waits for a certain delay (integrating time);
   (c) Then 50 lines are read using the normal readout process.

   At each iteration, the delay is increased.

The resulting frame will present several blocks 50 lines, each one exhibiting a similar gradient, but with a increasing illumination level. The gradient is the same for each block, and by subtracting it, only remains the flux integrated during the delay (which increases for each block).

As the delay increases, we are then able to study the CCD response at very low fluxes, with an effective exposure time controlled at the nanosecond level, something impossible to do with a mechanical shutter. A frame obtained that way is show on fig 3.1.

This sequence takes advantage of the advanced features of the sequencer language (`SET` and `WHILE` meta-instructions).

```
[...]
[functions]
    [...]

    RevTransferLine:  # Single line reverse transfer
      clocks:          P1, P2, P3, P4
      slices:
         BufferP      = 0,  1,  1,  0
         OverlapP     = 1,  1,  1,  0
         TimeP        = 1,  1,  0,  0
         OverlapP     = 1,  1,  0,  1
         TimeP        = 1,  0,  0,  1
         OverlapP     = 1,  0,  1,  1
         TimeP        = 0,  0,  1,  1
         OverlapP     = 0,  1,  1,  1
         TimeP        = 0,  1,  1,  0
         5000 ns      = 0,  1,  1,  0   # to match e2v timing
      constants:      S1=1, S2=1

[...]
[subroutines]
    [...]

    LinearityFrame:  # Special
        JSR      FlushRegister
        CALL     StartOfImage
        SET      uplines     100
        SET      downlines   50
        SET      wait      0
        SET      rep       0
        SET      maxrep   20
        WHILE    rep < maxrep    DO
            CALL     RevTransferLine   repeat(uplines)
            JSR      Exposure25ms      repeat(wait)
            JSR      WindowLine        repeat(downlines)
            SET      rep      rep + 1
            SET      wait     wait + 10
        DONE
        CALL     EndOfImage
        RTS

[mains]
    [...]

    LinearityAcquisition:   # Special
        JSR      ClearCCD        repeat(@CleaningNumber)
        JSR      LinearityFrame
        END
```

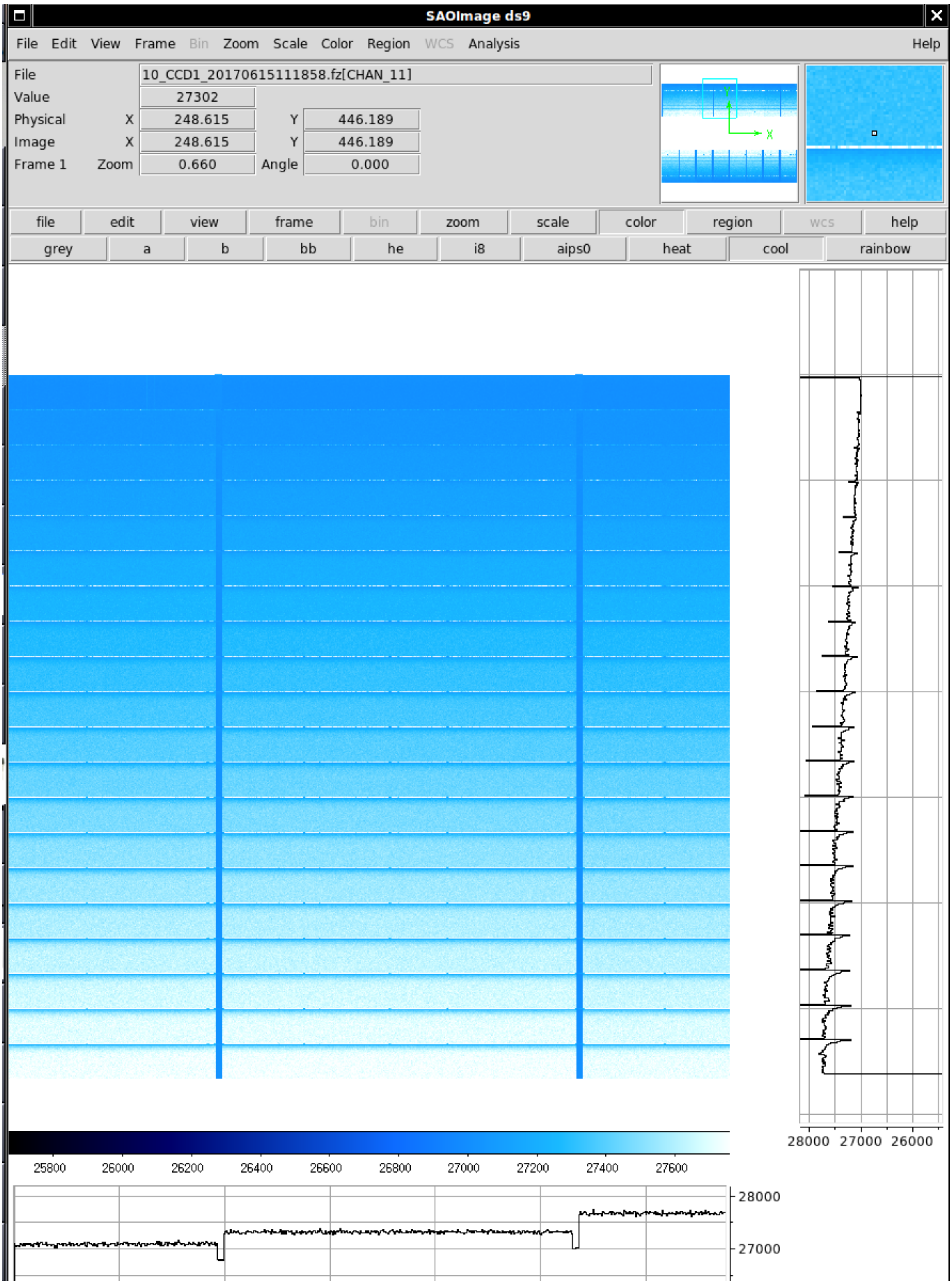

Figure 3.1: CCD frame obtained by alterning between reverse parallel transfer (100 lines) and normal parallel transfer (50 lines, with pixel readout), and an increasing delay between reverse and normal parallel transfers at each iteration. Such type of sequencer program may be used to measure non linearity at very low illumination.

4

Sequencer program compilation

4.1 Available compilers and tools

4.1.1 CCS java classes

TBW (CCS classes involved into the sequencer compilation process: E. Aubourg)

4.1.2 Standalone python tools

As a development toolbox, we provide the sequencer compiler written in python, as a very light standalone python package, available here:

`http://supernovae.in2p3.fr/~llg/LSST/REB/sequencer/lsst-sequencer-compiler-0.8.tar.bz2`

This package provides a python program named `seqcompiler`, which transforms a sequencer program (`*.txt` or `*.seq`) into its compiled version (`*.compiled`, see below), ready to be loaded into the REB FPGA program memory.

4.1.3 Low level acquisition tools

Once the sequencer program has been compiled, the program may be loaded into the program memory of the REB FPGA by successive calls to `rms_read`, a tool from the LSST acquisition system. In a similar way, a dump of the entire program memory may be done by successive calls to `rms_write`.

4.2 Compilation process

The compilation of a sequencer program file follows these steps:

1. The main sequencer file and all included files, are recursively parsed. Later definitions replace previous one.

2. Functions are encoded, with time slices durations and outputs properly formatted;

3. subroutines and main programs are compiled, in their definition order. Pseudo-instructions `SET`, `IF`, and `WHILE` are processed. The resulting code is translated in elementary instructions `CALL`, `JSR`, `RTS` and `END`. At this step, global and local parameters are replaced by their values.

4. Functions names are replaced by functions ids.

5. subroutines and main programs are relocated in the program memory. subroutines and main programs names are replaced by their address value.

6. Instructions are translated into opcodes, ready to be loaded into the FPGA program memory.

The end products of the compilation process are:

- A list of pairs address/value to load into the REB FPGA program memory;
- A list of main programs (mains) addresses, with their symbolic names (e.g. `Bias`, `Acquisition`);
- A list of (modifiable) pointers with their symbolic name, their address and their initial value.

4.3 Compiled sequencer programs: file format

In this section, we described the file format for compiled sequencer programs (`*.compiled`). This file format may also be used for fast loading of a sequencer program, and for dumps of the REB FPGA program memory.

The file format is quite simple: it mainly consists of a list of memory addresses and values to be affected at the corresponding adresses, like this:

```
[...]
0x100010: 0x00000630
0x100011: 0x00000e30
0x100012: 0x00000c30
0x100013: 0x00000d30
0x100014: 0x00000930
0x100015: 0x00000b30
0x100016: 0x00000330
0x100017: 0x00000730
0x100018: 0x00000630
0x100019: 0x00000630
0x10001a: 0x00000000
[...]
```

This file format is very simple to parse and process, and loading the program into the REB FPGA program memory may be done by successive calls to `rms_write` (see above).

Extra informations are provided as *comments* (prefixed by a '#'). For each elementary function, the name and the execution time are given in commented lines. The list of pointers and the list of available mains/subroutines are provided in a commented section as well.

The symbolic name and relative address of each `main` are given with the following syntax:

```
[...]
## =====================================================
# [subroutines/mains]
##
## -----------------------------------------------------
## Main/Subroutine relative addresses
## (program base addr 0x300000)
##
# WindowLine: 0x000048
# AcquireFrame: 0x000088
```

```
# PixelFlushLine: 0x000040
# InfiniteWait: 0x000028
# Exposure25ms: 0x000078
# Clear: 0x000008
# FakeFrame: 0x000070
# ReadFrame: 0x000068
# RawBias: 0x000000
# Dark: 0x000030
# Bias: 0x000010
# CloseShutter: 0x000058
# FlushRegister: 0x000060
# NoAcquisition: 0x000020
# ClearCCD: 0x000080
# FlushLine: 0x000038
# Acquisition: 0x000018
# WindowWithOverscan: 0x000050
## -------------------------------------------------------
[...]
```

And for the pointers, the symbolic name and the pointer type are given as comments at the end of each line:

```
[...]
## =======================================================
# [pointers]
##
0x380000: 0x000050   # REP_SUBR:  ExposureTime
0x380004: 0x000002   # REP_SUBR:  CleaningNumber
0x340000: 0x000000   # MAIN:  Main
0x380002: 0x000032   # REP_SUBR:  ReadRows
0x370000: 0x000038   # PTR_SUBR:  CleaningSubr
0x380005: 0x000064   # REP_SUBR:  FlushLines
0x360004: 0x00c350   # REP_FUNC:  ShutterTime
0x360000: 0x00012c   # REP_FUNC:  PreCols
0x360003: 0x000032   # REP_FUNC:  OverCols
0x360005: 0x00c350   # REP_FUNC:  FlushTime
0x380003: 0x0003ca   # REP_SUBR:  PostRows
0x380001: 0x0003e8   # REP_SUBR:  PreRows
0x360002: 0x0000e2   # REP_FUNC:  PostCols
0x350001: 0x000006   # PTR_FUNC:  ClosingFunc
0x360001: 0x000032   # REP_FUNC:  ReadCols
0x350000: 0x000007   # PTR_FUNC:  Exposure
## =======================================================
[...]
```

4.4 Compiled program: complete example

We provide here the complete compiled file as an example to illustrate the file format.

```
## LSST REB compiled sequencer file
## REB: REB5
## Source: seq-overp-full.seq
## Compilation date: 2017-04-19 14:36:08.847949
```

```
## Compiler: python lsst-seq-compiler 0.9
## Compiler authors: L. Le Guillou, C. Juramy
## ======================================================
# [functions]
##
## ------------------------------------------------------
## function: #0
##   name:  Default
##   description:  Default state when not operating
##   execution time:  102
##
0x100000: 0x000006bc
0x100001: 0x00000000
0x100002: 0x00000000
0x100003: 0x00000000
0x100004: 0x00000000
0x100005: 0x00000000
0x100006: 0x00000000
0x100007: 0x00000000
0x100008: 0x00000000
0x100009: 0x00000000
0x10000a: 0x00000000
0x10000b: 0x00000000
0x10000c: 0x00000000
0x10000d: 0x00000000
0x10000e: 0x00000000
0x10000f: 0x00000000
0x200000: 0x00000063
0x200001: 0x00000000
0x200002: 0x00000000
0x200003: 0x00000000
0x200004: 0x00000000
0x200005: 0x00000000
0x200006: 0x00000000
0x200007: 0x00000000
0x200008: 0x00000000
0x200009: 0x00000000
0x20000a: 0x00000000
0x20000b: 0x00000000
0x20000c: 0x00000000
0x20000d: 0x00000000
0x20000e: 0x00000000
0x20000f: 0x00000000
## ------------------------------------------------------
## function: #1
##   name:  TransferLine
##   description:  Single line transfer
##   execution time:  3004
##
0x100010: 0x00000630
0x100011: 0x00000e30
0x100012: 0x00000c30
0x100013: 0x00000d30
0x100014: 0x00000930
0x100015: 0x00000b30
0x100016: 0x00000330
0x100017: 0x00000730
0x100018: 0x00000630
0x100019: 0x00000630
0x10001a: 0x00000000
0x10001b: 0x00000000
0x10001c: 0x00000000
0x10001d: 0x00000000
0x10001e: 0x00000000
0x10001f: 0x00000000
0x200010: 0x000000f9
0x200011: 0x00000064
0x200012: 0x000001f4
0x200013: 0x00000064
0x200014: 0x000001f4
0x200015: 0x00000064
0x200016: 0x000001f4
0x200017: 0x00000064
0x200018: 0x00000064
0x200019: 0x000002f0
0x20001a: 0x00000000
```

```
0x20001b: 0x00000000
0x20001c: 0x00000000
0x20001d: 0x00000000
0x20001e: 0x00000000
0x20001f: 0x00000000
## -------------------------------------------------------
## function: #2
##   name:  ParallelFlush
##   description:  Single line transfer with all serial register clocks high to flush it
##   execution time:  10000
##
0x100020: 0x000006f4
0x100021: 0x00000ef4
0x100022: 0x00000cf4
0x100023: 0x00000df4
0x100024: 0x000009f4
0x100025: 0x00000bf4
0x100026: 0x000003f4
0x100027: 0x000007f4
0x100028: 0x000006f4
0x100029: 0x000006f4
0x10002a: 0x00000000
0x10002b: 0x00000000
0x10002c: 0x00000000
0x10002d: 0x00000000
0x10002e: 0x00000000
0x10002f: 0x00000000
0x200020: 0x000002ed
0x200021: 0x00000064
0x200022: 0x000005dc
0x200023: 0x00000064
0x200024: 0x000005dc
0x200025: 0x00000064
0x200026: 0x000005dc
0x200027: 0x00000064
0x200028: 0x00000e10
0x200029: 0x000002ec
0x20002a: 0x00000000
0x20002b: 0x00000000
0x20002c: 0x00000000
0x20002d: 0x00000000
0x20002e: 0x00000000
0x20002f: 0x00000000
## -------------------------------------------------------
## function: #3
##   name:  ReadPixel
##   description:  Single pixel read
##   execution time:  186
##
0x100030: 0x000016a0
0x100031: 0x000006a0
0x100032: 0x000006e4
0x100033: 0x00000644
0x100034: 0x0000064c
0x100035: 0x00000640
0x100036: 0x00000642
0x100037: 0x00000650
0x100038: 0x00000610
0x100039: 0x00000611
0x10003a: 0x00000630
0x10003b: 0x00000000
0x10003c: 0x00000000
0x10003d: 0x00000000
0x10003e: 0x00000000
0x10003f: 0x00000000
0x200030: 0x00000004
0x200031: 0x0000000f
0x200032: 0x00000008
0x200033: 0x00000008
0x200034: 0x00000019
0x200035: 0x0000000d
0x200036: 0x00000020
0x200037: 0x00000008
0x200038: 0x00000020
0x200039: 0x00000020
0x20003a: 0x00000006
```

```
0x20003b: 0x00000000
0x20003c: 0x00000000
0x20003d: 0x00000000
0x20003e: 0x00000000
0x20003f: 0x00000000
## ------------------------------------------------------
## function: #4
##   name:  StartOfImage
##   description:  Signals start of frame to be recorded
##   execution time:  180
##
0x100040: 0x000006b0
0x100041: 0x000026b0
0x100042: 0x000006b0
0x100043: 0x00000000
0x100044: 0x00000000
0x100045: 0x00000000
0x100046: 0x00000000
0x100047: 0x00000000
0x100048: 0x00000000
0x100049: 0x00000000
0x10004a: 0x00000000
0x10004b: 0x00000000
0x10004c: 0x00000000
0x10004d: 0x00000000
0x10004e: 0x00000000
0x10004f: 0x00000000
0x200040: 0x0000009f
0x200041: 0x0000000a
0x200042: 0x00000008
0x200043: 0x00000000
0x200044: 0x00000000
0x200045: 0x00000000
0x200046: 0x00000000
0x200047: 0x00000000
0x200048: 0x00000000
0x200049: 0x00000000
0x20004a: 0x00000000
0x20004b: 0x00000000
0x20004c: 0x00000000
0x20004d: 0x00000000
0x20004e: 0x00000000
0x20004f: 0x00000000
## ------------------------------------------------------
## function: #5
##   name:  EndOfImage
##   description:  Signals end of frame to be recorded
##   execution time:  180
##
0x100050: 0x000006b0
0x100051: 0x000046b0
0x100052: 0x000006b0
0x100053: 0x00000000
0x100054: 0x00000000
0x100055: 0x00000000
0x100056: 0x00000000
0x100057: 0x00000000
0x100058: 0x00000000
0x100059: 0x00000000
0x10005a: 0x00000000
0x10005b: 0x00000000
0x10005c: 0x00000000
0x10005d: 0x00000000
0x10005e: 0x00000000
0x10005f: 0x00000000
0x200050: 0x0000009f
0x200051: 0x0000000a
0x200052: 0x00000008
0x200053: 0x00000000
0x200054: 0x00000000
0x200055: 0x00000000
0x200056: 0x00000000
0x200057: 0x00000000
0x200058: 0x00000000
0x200059: 0x00000000
0x20005a: 0x00000000
```

```
0x20005b: 0x00000000
0x20005c: 0x00000000
0x20005d: 0x00000000
0x20005e: 0x00000000
0x20005f: 0x00000000
## ------------------------------------------------------
## function: #6
##   name:  SerialFlush
##   description:  Single pixel flush with timing set by FlushS parameter
##   execution time:  186
##
0x100060: 0x000006a4
0x100061: 0x000006e4
0x100062: 0x00000644
0x100063: 0x00000654
0x100064: 0x00000614
0x100065: 0x00000634
0x100066: 0x00000000
0x100067: 0x00000000
0x100068: 0x00000000
0x100069: 0x00000000
0x10006a: 0x00000000
0x10006b: 0x00000000
0x10006c: 0x00000000
0x10006d: 0x00000000
0x10006e: 0x00000000
0x10006f: 0x00000000
0x200060: 0x00000035
0x200061: 0x00000008
0x200062: 0x00000036
0x200063: 0x00000008
0x200064: 0x00000036
0x200065: 0x00000006
0x200066: 0x00000000
0x200067: 0x00000000
0x200068: 0x00000000
0x200069: 0x00000000
0x20006a: 0x00000000
0x20006b: 0x00000000
0x20006c: 0x00000000
0x20006d: 0x00000000
0x20006e: 0x00000000
0x20006f: 0x00000000
## ------------------------------------------------------
## function: #7
##   name:  ExposureFlush
##   description:  Exposure while flushing serial register (testing only), same timing as SerialFlushReg
##   execution time:  186
##
0x100070: 0x000106a4
0x100071: 0x000106e4
0x100072: 0x000106c4
0x100073: 0x000106d4
0x100074: 0x00010694
0x100075: 0x000106b4
0x100076: 0x00000000
0x100077: 0x00000000
0x100078: 0x00000000
0x100079: 0x00000000
0x10007a: 0x00000000
0x10007b: 0x00000000
0x10007c: 0x00000000
0x10007d: 0x00000000
0x10007e: 0x00000000
0x10007f: 0x00000000
0x200070: 0x00000035
0x200071: 0x00000008
0x200072: 0x00000036
0x200073: 0x00000008
0x200074: 0x00000036
0x200075: 0x00000006
0x200076: 0x00000000
0x200077: 0x00000000
0x200078: 0x00000000
0x200079: 0x00000000
0x20007a: 0x00000000
```

```
0x20007b: 0x00000000
0x20007c: 0x00000000
0x20007d: 0x00000000
0x20007e: 0x00000000
0x20007f: 0x00000000
## ------------------------------------------------------
## function: #8
##   name:  DarkNoFlush
##   description:  Dark without flushing serial register, same timing as SerialFlushReg
##   execution time:  186
##
0x100080: 0x000006f4
0x100081: 0x000006f4
0x100082: 0x000006f4
0x100083: 0x000006f4
0x100084: 0x000006f4
0x100085: 0x000006f4
0x100086: 0x00000000
0x100087: 0x00000000
0x100088: 0x00000000
0x100089: 0x00000000
0x10008a: 0x00000000
0x10008b: 0x00000000
0x10008c: 0x00000000
0x10008d: 0x00000000
0x10008e: 0x00000000
0x10008f: 0x00000000
0x200080: 0x00000035
0x200081: 0x00000008
0x200082: 0x00000036
0x200083: 0x00000008
0x200084: 0x00000036
0x200085: 0x00000006
0x200086: 0x00000000
0x200087: 0x00000000
0x200088: 0x00000000
0x200089: 0x00000000
0x20008a: 0x00000000
0x20008b: 0x00000000
0x20008c: 0x00000000
0x20008d: 0x00000000
0x20008e: 0x00000000
0x20008f: 0x00000000
## ------------------------------------------------------
## function: #9
##   name:  ExposureNoFlush
##   description:  Exposure without flushing serial register (testing only), same timing as SerialFlushReg
##   execution time:  186
##
0x100090: 0x000106f4
0x100091: 0x000106f4
0x100092: 0x000106f4
0x100093: 0x000106f4
0x100094: 0x000106f4
0x100095: 0x000106f4
0x100096: 0x00000000
0x100097: 0x00000000
0x100098: 0x00000000
0x100099: 0x00000000
0x10009a: 0x00000000
0x10009b: 0x00000000
0x10009c: 0x00000000
0x10009d: 0x00000000
0x10009e: 0x00000000
0x10009f: 0x00000000
0x200090: 0x00000035
0x200091: 0x00000008
0x200092: 0x00000036
0x200093: 0x00000008
0x200094: 0x00000036
0x200095: 0x00000006
0x200096: 0x00000000
0x200097: 0x00000000
0x200098: 0x00000000
0x200099: 0x00000000
0x20009a: 0x00000000
```

```
0x20009b: 0x00000000
0x20009c: 0x00000000
0x20009d: 0x00000000
0x20009e: 0x00000000
0x20009f: 0x00000000
## -----------------------------------------------------
## function: #10
##   name:  SlowFlush
##   description:  Simultaneous serial and parallel flush, slow (waiting pattern)
##   execution time:  13000
##
0x1000a0: 0x000006ac
0x1000a1: 0x0000064c
0x1000a2: 0x0000061c
0x1000a3: 0x0000062c
0x1000a4: 0x0000064c
0x1000a5: 0x0000061c
0x1000a6: 0x0000062c
0x1000a7: 0x0000064c
0x1000a8: 0x0000061c
0x1000a9: 0x000006ac
0x1000aa: 0x00000cac
0x1000ab: 0x000009ac
0x1000ac: 0x000003ac
0x1000ad: 0x0000062c
0x1000ae: 0x00000000
0x1000af: 0x00000000
0x2000a0: 0x000001f3
0x2000a1: 0x000001f4
0x2000a2: 0x000001f4
0x2000a3: 0x000001f4
0x2000a4: 0x000001f4
0x2000a5: 0x000001f4
0x2000a6: 0x000001f4
0x2000a7: 0x000001f4
0x2000a8: 0x000001f4
0x2000a9: 0x000001f4
0x2000aa: 0x000007d0
0x2000ab: 0x000007d0
0x2000ac: 0x000007d0
0x2000ad: 0x000007ce
0x2000ae: 0x00000000
0x2000af: 0x00000000
## =====================================================
# [subroutines/mains]
##
## -----------------------------------------------------
## Main/Subroutine relative addresses
## (program base addr 0x300000)
##
# WindowLine: 0x000048
# AcquireFrame: 0x000088
# PixelFlushLine: 0x000040
# InfiniteWait: 0x000028
# Exposure25ms: 0x000078
# Clear: 0x000008
# FakeFrame: 0x000070
# ReadFrame: 0x000068
# RawBias: 0x000000
# Dark: 0x000030
# Bias: 0x000010
# CloseShutter: 0x000058
# FlushRegister: 0x000060
# NoAcquisition: 0x000020
# ClearCCD: 0x000080
# FlushLine: 0x000038
# Acquisition: 0x000018
# WindowWithOverscan: 0x000050
## -----------------------------------------------------
0x300000: 0x50680001
0x300001: 0xf0000000
0x300008: 0x70800004
0x300009: 0xf0000000
0x300010: 0x70800004
0x300011: 0x50680001
0x300012: 0xf0000000
```

```
0x300018: 0x50880001
0x300019: 0x50680001
0x30001a: 0xf0000000
0x300020: 0x50880001
0x300021: 0x50700001
0x300022: 0xf0000000
0x300028: 0x1a800000
0x300029: 0xf0000000
0x300030: 0x50880001
0x300031: 0x50680001
0x300032: 0xf0000000
0x300038: 0x12000001
0x300039: 0xe0000000
0x300040: 0x11000001
0x300041: 0x16000240
0x300042: 0xe0000000
0x300048: 0x11000001
0x300049: 0x36000000
0x30004a: 0x33000001
0x30004b: 0x36000002
0x30004c: 0xe0000000
0x300050: 0x11000001
0x300051: 0x36000000
0x300052: 0x33000001
0x300053: 0x36000002
0x300054: 0x33000003
0x300055: 0xe0000000
0x300058: 0x41000004
0x300059: 0xe0000000
0x300060: 0x36000005
0x300061: 0xe0000000
0x300068: 0x70380001
0x300069: 0x50600001
0x30006a: 0x14000001
0x30006b: 0x70480002
0x30006c: 0x15000001
0x30006d: 0x70380003
0x30006e: 0xe0000000
0x300070: 0x70380001
0x300071: 0x50600001
0x300072: 0x70480002
0x300073: 0x70380003
0x300074: 0xe0000000
0x300078: 0x20003481
0x300079: 0xe0000000
0x300080: 0x60000800
0x300081: 0xe0000000
0x300088: 0x70800004
0x300089: 0x70780000
0x30008a: 0x50580001
0x30008b: 0xe0000000
## =======================================================
# [pointers]
##
0x380000: 0x000050    # REP_SUBR:  ExposureTime
0x380004: 0x000002    # REP_SUBR:  CleaningNumber
0x340000: 0x000000    # MAIN:  Main
0x380002: 0x000032    # REP_SUBR:  ReadRows
0x370000: 0x000038    # PTR_SUBR:  CleaningSubr
0x380005: 0x000064    # REP_SUBR:  FlushLines
0x360004: 0x00c350    # REP_FUNC:  ShutterTime
0x360000: 0x00012c    # REP_FUNC:  PreCols
0x360003: 0x000032    # REP_FUNC:  OverCols
0x360005: 0x00c350    # REP_FUNC:  FlushTime
0x380003: 0x0003ca    # REP_SUBR:  PostRows
0x380001: 0x0003e8    # REP_SUBR:  PreRows
0x360002: 0x0000e2    # REP_FUNC:  PostCols
0x350001: 0x000006    # PTR_FUNC:  ClosingFunc
0x360001: 0x000032    # REP_FUNC:  ReadCols
0x350000: 0x000007    # PTR_FUNC:  Exposure
## =======================================================
```

5

Language grammar

In this section we provide the formal grammar of the sequencer language in the Backus–Naur form (also known as the "BNF" format).

5.1 Basic language elements

```
ZMSP ::= [ \t]* # zero or more spaces
OMSP ::= [ \t]+ # one or more spaces
NEWLINE ::= (\n | \r\n | \r) # newline
COMMENT ::= '\#' .*   # [until NEWLINE]
INTEGER ::= [0-9]+
ADDRESS ::= ((0x)?[0-9a-f]+)
NAME ::= [A-Za-z][0-9A-Za-z\_]*

DURATION_UNIT ::= ( 'ns' | 'us' | 'ms' | 's' )
DURATION_VALUE ::= INTEGER SPACE* DURATION_UNIT

FILE ::= [0-9A-Za-z\_\-\.\\\\]*

EMPTY_LINE ::= SPACE* COMMENT? NEWLINE
```

5.2 Mathematical expressions

Only a small subset of operators (addition, subtraction, multiplication and comparisons) are accepted.

```
CONSTANT_NAME ::= NAME

EXPR_CMP ::= ( EXPR_ADD SPACE*
               ( '==' | '!=' | '<=' | '<' | '>=' | '>' )
               SPACE* EXPR_ADD ) | EXPR_ADD

EXPR_ADD ::= EXPR_MUL  ( SPACE* ( '+' | '-' ) SPACE* EXPR_MUL)*

EXPR_MUL ::= EXPR_ATOM ( SPACE* '*' SPACE* EXPR_ATOM)*

EXPR_ATOM ::= '(' SPACE* EXPR_ADD SPACE* ')' |
              INTEGER | CONSTANT_NAME
```

```
EXPRESSION ::= EXPR_CMP
```

5.3 Included files

```
FILE_NAME ::= FILE

INCLUDE_DEF_LINE ::= SPACE* FILE_NAME SPACE* COMMENT? NEWLINE

INCLUDE_SECTION_MARKER ::= '[includes]' SPACE* COMMENT? NEWLINE

INCLUDE_SECTION ::=
  INCLUDE_SECTION_MARKER ( EMPTY_LINE | INCLUDE_DEF_LINE )*
```

5.4 Global parameters ("constants")

```
CONSTANT_NAME ::= NAME

CONSTANT_VALUE ::= DURATION_VALUE | EXPRESSION

CONSTANT_DEF_LINE ::=
  SPACE* CONSTANT_NAME SPACE* ':'
  CONSTANT_VALUE SPACE* COMMENT? NEWLINE

CONSTANT_SECTION_MARKER ::= '[constants]' SPACE* COMMENT? NEWLINE

CONSTANT_SECTION ::=
  CONSTANT_SECTION_MARKER ( EMPTY_LINE | CONSTANT_DEF_LINE )*
```

5.5 Clock lines

```
CLOCK_NAME ::= NAME

CLOCK_ID ::= INTEGER

CLOCK_DEF_LINE ::=
  SPACE* CLOCK_NAME SPACE* ':' CLOCK_ID SPACE* COMMENT? NEWLINE

CLOCK_SECTION_MARKER ::= '[clocks]' SPACE* COMMENT? NEWLINE

CLOCK_SECTION ::=
  CLOCK_SECTION_MARKER ( EMPTY_LINE | CLOCK_DEF_LINE )*
```

5.6 Indirect addressing ("pointers")

```
REP_FUNC_NAME ::= NAME
REP_SUBR_NAME ::= NAME
PTR_FUNC_NAME ::= NAME
PTR_SUBR_NAME ::= NAME
```

```
REP_FUNC_DEF_LINE ::=
  SPACE* 'REP_FUNC' SPACE+ REP_FUNC_NAME
  SPACE+ EXPR SPACE* COMMENT? NEWLINE

REP_SUBR_DEF_LINE ::=
  SPACE* 'REP_SUBR' SPACE+ REP_SUBR_NAME
  SPACE+ EXPR SPACE* COMMENT? NEWLINE

PTR_FUNC_DEF_LINE ::=
  SPACE* 'PTR_FUNC' SPACE+ PTR_FUNC_NAME
  SPACE+ (FUNC_NAME | FUNC_ID) SPACE* COMMENT? NEWLINE

PTR_SUBR_DEF_LINE ::=
  SPACE* 'PTR_SUBR' SPACE+ PTR_SUBR_NAME
  SPACE+ (SUBR_NAME | ADDRESS) SPACE* COMMENT? NEWLINE

MAIN_DEF_LINE ::=
  SPACE* 'MAIN' SPACE+ PTR_SUBR_NAME
  SPACE+ (SUBR_NAME | ADDRESS) SPACE* COMMENT? NEWLINE

PTR_DEF_LINE ::=  ( REP_FUNC_DEF_LINE | REP_SUBR_DEF_LINE |
                    PTR_FUNC_DEF_LINE | PTR_SUBR_DEF_LINE |
                    MAIN_DEF_LINE )

PTR_SECTION_MARKER ::= '[pointers]' SPACE* COMMENT? NEWLINE

PTR_SECTION ::=
  PTR_SECTION_MARKER ( EMPTY_LINE | PTR_DEF_LINE )*
```

5.7 Sequencer functions

```
FUNC_NAME ::= NAME

FUNC_ID ::= INTEGER

FUNC_NAME_DEF_LINE ::=
  SPACE* FUNC_NAME SPACE* ':' SPACE* COMMENT? NEWLINE

FUNC_CLOCKS_MARKER ::= 'clocks'

FUNC_CLOCKS_NAMES_LINE ::=
  SPACE* FUNC_CLOCKS_MARKER SPACE* ':'
  SPACE* CLOCK_NAME SPACE* (',' SPACE* CLOCK_NAME SPACE*)*

FUNC_SLICES_MARKER ::= 'slices'

FUNC_SLICES_MARKER_LINE ::=
  SPACE* FUNC_SLICES_MARKER SPACE* ':' SPACE* COMMENT? NEWLINE

FUNC_SLICE_DEF_LINE ::=
  SPACE* ( DURATION_VALUE | CONSTANT_NAME ) SPACE* '='
  SPACE* ( '0' | '1' ) SPACE* (',' SPACE* ( '0' | '1' ) SPACE* )*
```

```
  SPACE* COMMENT? NEWLINE

FUNC_SLICES_DEFS_BLOCK ::=
  FUNC_SLICES_MARKER_LINE
  EMPTY_LINE*
  FUNC_SLICE_DEF_LINE
  ( FUNC_SLICE_DEF_LINE | EMPTY_LINE )*

FUNC_CONSTANTS_MARKER ::= 'constants'

FUNC_CONSTANTS_DEFS_LINE ::=
  SPACE* FUNC_CONSTANTS_MARKER SPACE* ':'
  SPACE* CLOCK_NAME SPACE* '=' SPACE* ( '0' | '1' )
  (',' SPACE* CLOCK_NAME SPACE* '=' SPACE* ( '0' | '1' ) )*
  SPACE* COMMENT? NEWLINE

FUNC_DEF_BLOCK ::=
  FUNC_NAME_DEF_LINE
  EMPTY_LINE*
  FUNC_CLOCKS_NAMES_LINE
  EMPTY_LINE*
  FUNC_SLICES_DEFS_BLOCK
  EMPTY_LINE*
  FUNC_CONSTANTS_DEFS_LINE?
  EMPTY_LINE*

FUNC_SECTION ::= FUNC_SECTION_MARKER FUNC_DEF_BLOCK*
```

5.8 Subroutines and main programs

```
SUBR_NAME ::= NAME

INSTR_CALL_LINE ::=
  SPACE* 'CALL' SPACE+
  ( ( '@' ( PTR_FUNC_NAME | ADDRESS) ) | FUNC_ID | FUNC_NAME )
  ( SPACE+ 'repeat(' SPACE*
    ( ( '@' ( REP_FUNC_NAME | ADDRESS ) ) |
      EXPRESSION | 'Inf' ) SPACE* ')' )
  SPACE* COMMENT? NEWLINE

INSTR_JSR_LINE ::=
  SPACE* 'JSR' SPACE+
  ( ( '@' ( PTR_SUBR_NAME | ADDRESS ) ) | ADDRESS | SUBR_NAME )
  ( SPACE+ 'repeat(' SPACE*
    ( ( '@' ( REP_SUBR_NAME | ADDRESS ) ) |
      EXPRESSION ) SPACE* ')' )
  SPACE* COMMENT? NEWLINE

INSTR_RTS_LINE ::=
  SPACE* 'RTS' SPACE* COMMENT? NEWLINE

INSTR_END_LINE ::=
  SPACE* 'END' SPACE* COMMENT? NEWLINE
```

```
INSTR_SET_BLOCK ::=
  SPACE* 'SET' SPACE+ CONSTANT_NAME
  SPACE+ EXPRESSION SPACE* COMMENT? NEWLINE

INSTR_WHILE_BEGIN ::=
  SPACE* 'WHILE' SPACE+ EXPRESSION SPACE+
  'DO' SPACE* COMMENT? NEWLINE

INSTR_WHILE_END ::=
  SPACE* 'DONE' SPACE* COMMENT? NEWLINE

INSTR_WHILE_BLOCK ::=
  INSTR_WHILE_BEGIN
  INSTR_BLOCK
  INSTR_WHILE_END

INSTR_IF_BEGIN ::=
  SPACE* 'IF' SPACE+ EXPRESSION SPACE+
  'THEN' SPACE* COMMENT? NEWLINE

INSTR_IF_END ::=
  SPACE* 'FI' SPACE* COMMENT? NEWLINE

INSTR_IF_BLOCK ::=
  INSTR_IF_BEGIN
  INSTR_BLOCK
  INSTR_IF_END

INSTR_BLOCK ::=
  ( INSTR_SET_BLOCK | INSTR_WHILE_BLOCK |
    INSTR_IF_BLOCK | INSTR_CALL_LINE |
    INSTR_JSR_LINE | EMPTY_LINE )*

SUBR_NAME_DEF_LINE ::=
  SPACE* SUBR_NAME SPACE* ':' SPACE* COMMENT? NEWLINE

SUBR_DEF_BLOCK ::=
  SUBR_NAME_DEF_LINE
  INSTR_BLOCK
  INSTR_RTS_LINE

MAIN_DEF_BLOCK ::=
  SUBR_NAME_DEF_LINE
  INSTR_BLOCK
  INSTR_END_LINE

SUBR_SECTION ::= SUBR_SECTION_MARKER SUBR_DEF_BLOCK*

MAIN_SECTION ::= MAIN_SECTION_MARKER MAIN_DEF_BLOCK*
```

5.9 Triggers

```
TRIGGER_ID  ::= INTEGER

TRIGGER_VALUE ::= SUBR_NAME

TRIGGER_DEF_LINE ::=
  SPACE* TRIGGER_ID SPACE* ':'
  SPACE* TRIGGER_VALUE SPACE* COMMENT? NEWLINE

TRIGGER_SECTION_MARKER ::= '[triggers]' SPACE* COMMENT? NEWLINE

TRIGGER_SECTION ::=
  TRIGGER_SECTION_MARKER ( EMPTY_LINE | TRIGGER_DEF_LINE )*
```

5.10 Sequencer file structure

```
SEQ ::=
    CONSTANT_SECTION
    CLOCK_SECTION
    PTR_SECTION?
    FUNC_SECTION
    SUBR_SECTION?
    MAIN_SECTION
    TRIGGER_SECTION?
```