

## NetXPTO - LinkPlanner

14 de Julho de 2017

---

## Conteúdo

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Simulator Structure</b>	<b>3</b>
2.1	System . . . . .	3
<b>3</b>	<b>Development Cycle</b>	<b>4</b>
<b>4</b>	<b>Visualizer</b>	<b>5</b>
<b>5</b>	<b>Case Studies</b>	<b>6</b>
5.1	QPSK Transmitter . . . . .	6
<b>6</b>	<b>Library</b>	<b>8</b>
6.1	Add . . . . .	9
6.2	Binary source . . . . .	10
6.3	Clock . . . . .	14
6.4	Decoder . . . . .	16
6.5	Discrete to continuous time . . . . .	19
6.6	Homodyne receiver . . . . .	21
6.7	IQ modulator . . . . .	25
6.8	Local Oscillator . . . . .	27
6.9	MQAM mapper . . . . .	29
6.10	MQAM transmitter . . . . .	32



LinkPlanner is a signals open-source simulator.

The major entity is the system.

A system comprises a set of blocks.

The blocks interact with each other through signals.

### 2.1 System

You can run the System

The NetXPTO-LinkPlanner has been developed by several people using git as a version control system. The NetXPTO-LinkPlanner repository is located in the GitHub site <http://github.com/netxpto/linkplanner>. The more updated functional version of the software is in the branch master. Master should be considered a functional beta version of the software. Periodically new releases are delivered from the master branch under the branch name Release<Year><Month><Day>. The integration of the work of all people is performed by Armando Nolasco Pinto in the branch Develop. Each developer has his own branch with his/her name.

visualizer

## 5.1 QPSK Transmitter

This system simulates a QPSK transmitter. A schematic representation of this system is shown in figure 5.1.

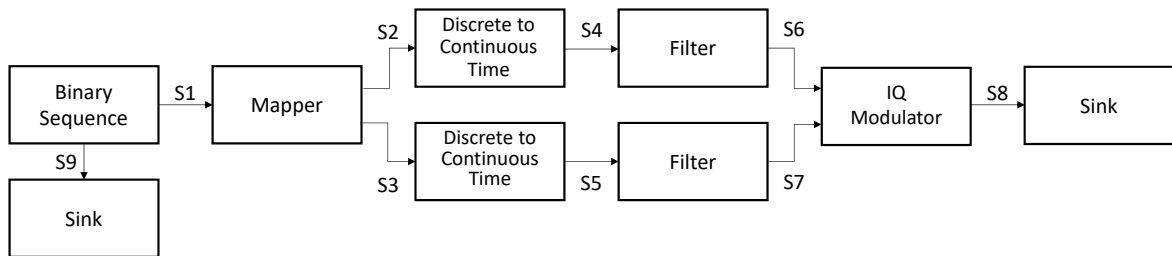


Figura 5.1: QPSK transmitter block diagram.

### System Input Parameters

**Parameter:** *sourceMode*

**Description:** Specifies the operation mode of the binary source.

**Accepted Values:** PseudoRandom, Random, DeterministicAppendZeros, DeterministicCyclic.

**Parameter:** *patternLength*

**Description:** Specifies the pattern length used by the source in the PseudoRandom mode.

**Accepted Values:** Integer between 1 and 32.

**Parameter:** *bitStream*

**Description:** Specifies the bit stream generated by the source in the DeterministicCyclic and DeterministicAppendZeros mode.

**Accepted Values:** "XXX..", where X is 0 or 1.

**Parameter:** *bitPeriod*

**Description:** Specifies the bit period, i.e. the inverse of the bit-rate.

**Accepted Values:** Any positive real value.

**Parameter:** *iqAmplitudes*

**Description:** Specifies the IQ amplitudes.

**Accepted Values:** Any four par of real values, for instance { { 1,1 }, { -1,1 }, { -1,-1 }, { 1,-1 } }, the first value correspond to the "00", the second to the "01", the third to the "10" and the forth to the "11".

**Parameter:** *numberOfBits*

**Description:** Specifies the number of bits generated by the binary source.

**Accepted Values:** Any positive integer value.

**Parameter:** *numberOfSamplesPerSymbol*

**Description:** Specifies the number of samples per symbol.

**Accepted Values:** Any positive integer value.

**Parameter:** *rollOffFactor*

**Description:** Specifies the roll off factor in the raised-cosine filter.

**Accepted Values:** A real value between 0 and 1.

**Parameter:** *impulseResponseTimeLength*

**Description:** Specifies the impulse response window time width in symbol periods.

**Accepted Values:** Any positive integer value.





## 6.1 Add

### Input Parameters

This block takes no parameters.

### Functional Description

This block accepts two signals and outputs one signal built from a sum of the two inputs. The input and output signals must be of the same type.

### Input Signals

**Number:** 2

**Type:** Real, Complex or Complex\_XY signal (ContinuousTimeContinuousAmplitude)

### Output Signals

**Number:** 1

**Type:** Real, Complex or Complex\_XY signal (ContinuousTimeContinuousAmplitude)

## 6.2 Binary source

This block generates a sequence of binary values (1 or 0) and it can work in four different modes:

- |                 |                             |
|-----------------|-----------------------------|
| 1. Random       | 3. DeterministicCyclic      |
| 2. PseudoRandom | 4. DeterministicAppendZeros |

This blocks doesn't accept any input signal. It produces any number of output signals.

### Input Parameters

**Parameter:** mode{PseudoRandom}  
(Random, PseudoRandom, DeterministicCyclic, DeterministicAppendZeros)

**Parameter:** probabilityOfZero{0.5}  
(real  $\in [0,1]$ )

**Parameter:** patternLength{7}  
(integer  $\in [1,32]$ )

**Parameter:** bitStream{"0100011101010101"}  
(string of 0's and 1's)

**Parameter:** numberOfBits{-1}  
(long int)

**Parameter:** bitPeriod{1.0/100e9}  
(double)

### Methods

BinarySource(vector<Signal \*> &InputSig, vector<Signal \*> &OutputSig) :Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setMode(BinarySourceMode m) BinarySourceMode const getMode(void)

void setProbabilityOfZero(double pZero)

double const getProbabilityOfZero(void)

void setBitStream(string bStream)

```

string const getBitStream(void)

void setNumberOfBits(long int nOfBits)

long int const getNumberOfBits(void)

void setPatternLength(int pLength)

int const getPatternLength(void)

void setBitPeriod(double bPeriod)

double const getBitPeriod(void)

```

### Functional description

The *mode* parameter allows the user to select between one of the four operation modes of the binary source.

**Random Mode** Generates a 0 with probability *probabilityOfZero* and a 1 with probability  $1 - \text{probabilityOfZero}$ .

**Pseudorandom Mode** Generates a pseudorandom sequence with period  $2^{\text{patternLength}} - 1$ .

**DeterministicCyclic Mode** Generates the sequence of 0's and 1's specified by *bitStream* and then repeats it.

**DeterministicAppendZeros Mode** Generates the sequence of 0's and 1's specified by *bitStream* and then it fills the rest of the buffer space with zeros.

### Input Signals

**Number:** 0

**Type:** Binary (DiscreteTimeDiscreteAmplitude)

### Output Signals

**Number:** 1 or more

**Type:** Binary (DiscreteTimeDiscreteAmplitude)

## Examples

### Random Mode

**PseudoRandom Mode** As an example consider a pseudorandom sequence with *patternLength*=3 which contains a total of 7 ( $2^3 - 1$ ) bits. In this sequence it is possible to find every combination of 0's and 1's that compose a 3 bit long subsequence with the exception of 000. For this example the possible subsequences are 010, 110, 101, 100, 111, 001 and 100 (they appear in figure 6.1 numbered in this order). Some of these require wrap.

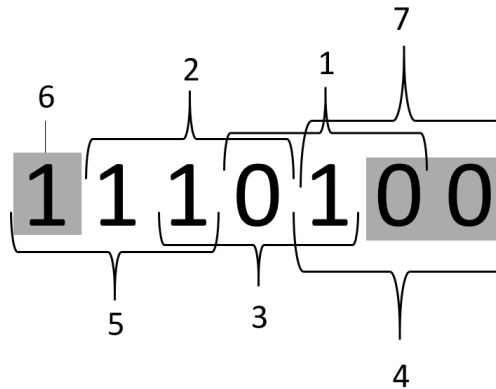


Figura 6.1: Example of a pseudorandom sequence with a pattern length equal to 3.

**DeterministicCyclic Mode** As an example take the *bit stream* '0100011101010101'. The generated binary signal is displayed in.

**DeterministicAppendZeros Mode** Take as an example the *bit stream* '0100011101010101'. The generated binary signal is displayed in 6.2.

### Sugestions for future improvement

Implement an input signal that can work as trigger.

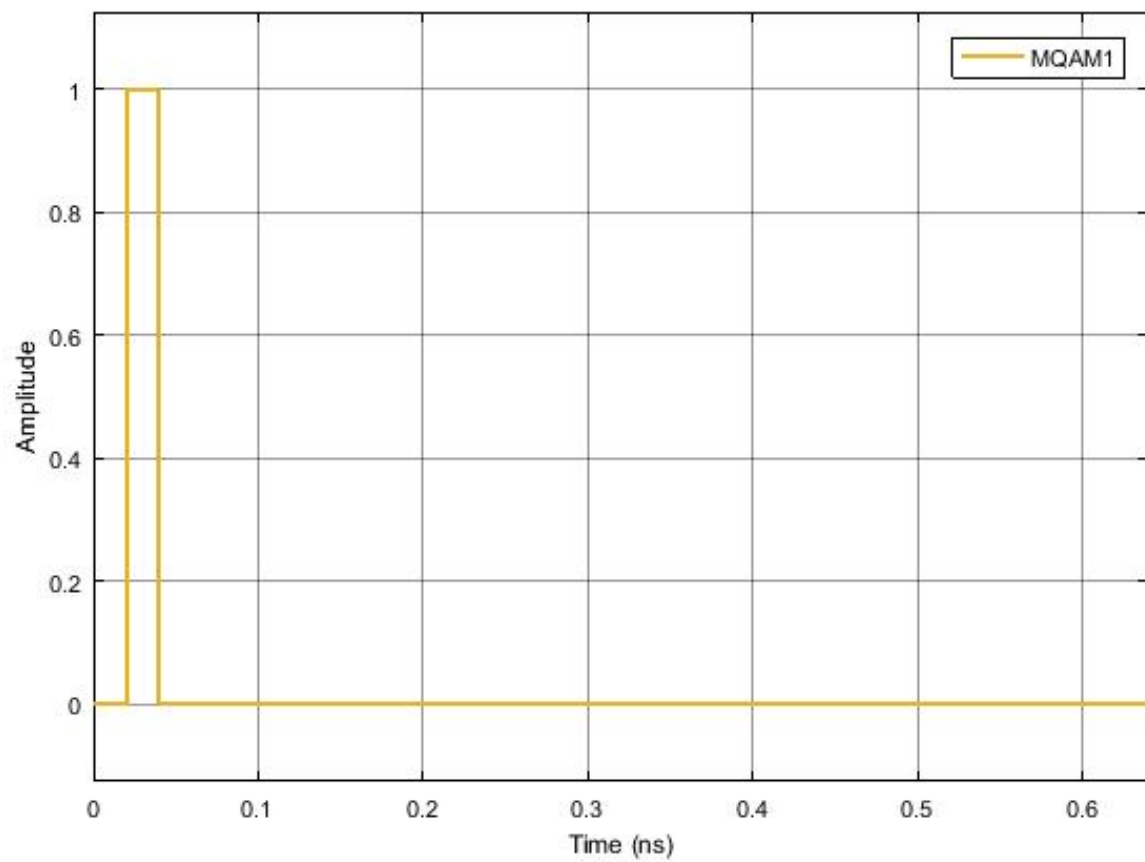


Figura 6.2: Binary signal generated by the block operating in the *Deterministic Append Zeros* mode with a binary sequence 01000...

## 6.3 Clock

This block doesn't accept any input signal. It outputs one signal that corresponds to a sequence of Dirac's delta functions with a user defined *period*.

### Input Parameters

**Parameter:** period{ 0.0 };

**Parameter:** samplingPeriod{ 0.0 };

### Methods

Clock()

Clock(vector<Signal \*> &InputSig, vector<Signal \*> &OutputSig) :Block(InputSig, OutputSig)

void initialize(void)

bool runBlock(void)

void setClockPeriod(double per)

void setSamplingPeriod(double sPeriod)

### Functional description

**Input Signals**

Number: 0

**Output Signals**

Number: 1

Type: Sequence of Dirac's delta functions.  
(TimeContinuousAmplitudeContinuousReal)

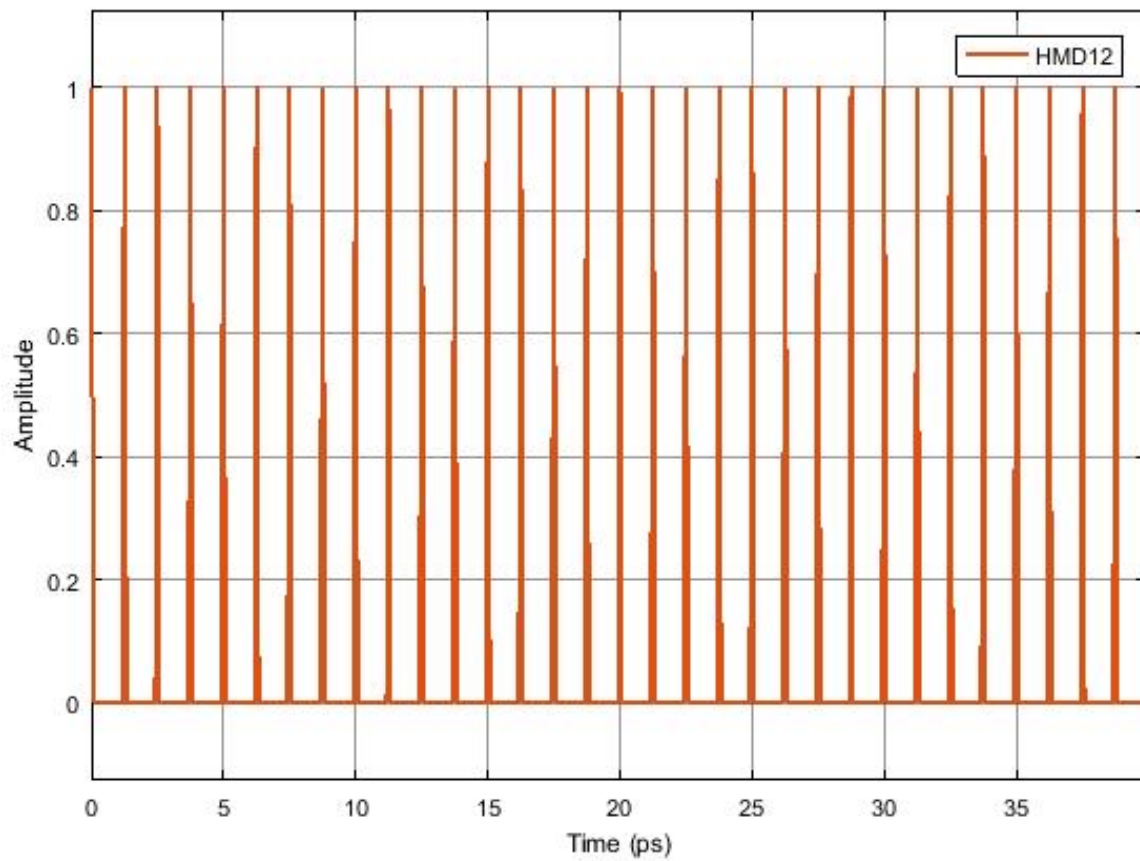
**Examples**

Figura 6.3: Example of the output signal of the clock

**Sugestions for future improvement**



## 6.4 Decoder

This block accepts a complex electrical signal and outputs a sequence of binary values (0's and 1's). Each point of the input signal corresponds to a pair of bits.

### Input Parameters

**Parameter:** `t_integer m{ 4 }`

**Parameter:** `vector<t_complex> iqAmplitudes{ { 1.0, 1.0 }, { -1.0, 1.0 }, { -1.0, -1.0 }, { 1.0, -1.0 } };`

### Methods

`Decoder()`

`Decoder(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig, OutputSig)`

`void initialize(void)`

`bool runBlock(void)`

`void setM(int mValue)`

`void getM()`

`void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)`

`vector<t_iqValues>getIqAmplitudes()`

### Functional description

This block makes the correspondence between a complex electrical signal and pair of binary values using a predetermined constellation.

To do so it computes the distance in the complex plane between each value of the input signal and each value of the *iqAmplitudes* vector selecting only the shortest one. It then converts the point in the IQ plane to a pair of bits making the correspondence between the input signal and a pair of bits.

## Input Signals

**Number:** 1

**Type:** Electrical complex (TimeContinuousAmplitudeContinuousReal)

## Output Signals

**Number:** 1

**Type:** Binary

## Examples

As an example take an input signal with positive real and imaginary parts. It would correspond to the first point of the *iqAmplitudes* vector and therefore it would be associated to the pair of bits 00.

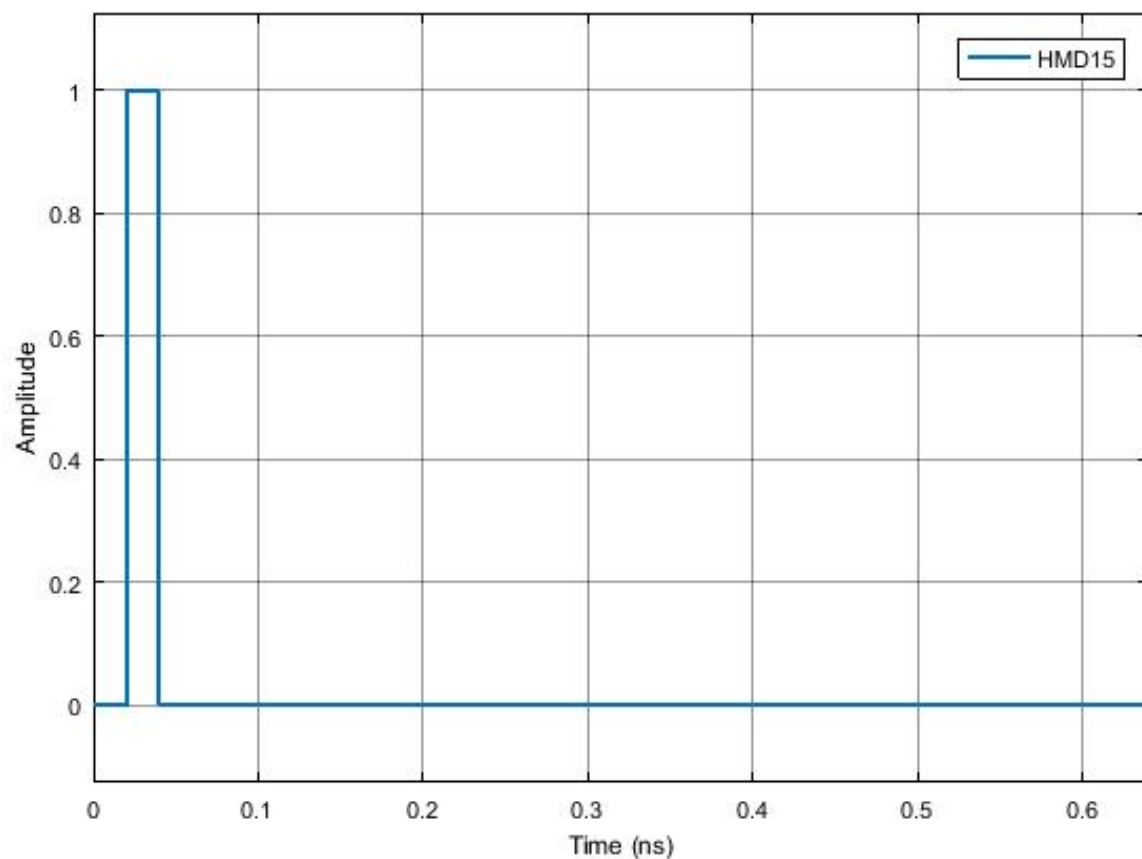


Figura 6.4: Example of the output signal of the decoder for a binary sequence 01. As expected it reproduces the initial bit stream

**Sugestions for future improvement**

## 6.5 Discrete to continuous time

This block converts a signal discrete in time to a signal continuous in time. It accepts one input signal that is a sequence of 1's and -1's and it produces one output signal that is a sequence of Dirac delta functions.

### Input Parameters

**Parameter:** numberOfSamplesPerSymbol{8}  
(int)

### Methods

```
DiscreteToContinuousTime(vector<Signal *> &inputSignals, vector<Signal *>
&outputSignals) :Block(inputSignals, outputSignals){};
```

```
void initialize(void);
```

```
bool runBlock(void);
```

```
void setNumberOfSamplesPerSymbol(int nSamplesPerSymbol)
```

```
int const getNumberOfSamplesPerSymbol(void)
```

### Functional Description

This block reads the input signal buffer value, puts it in the output signal buffer and it fills the rest of the space available for that symbol with zeros. The space available in the buffer for each symbol is given by the parameter *numberOfSamplesPerSymbol*.

### Input Signals

**Number** : 1

**Type** : Sequence of 1's and -1's. (DiscreteTimeDiscreteAmplitude)

### Output Signals

**Number** : 1

**Type** : Sequence of Dirac delta functions (ContinuousTimeDiscreteAmplitude)

### Example

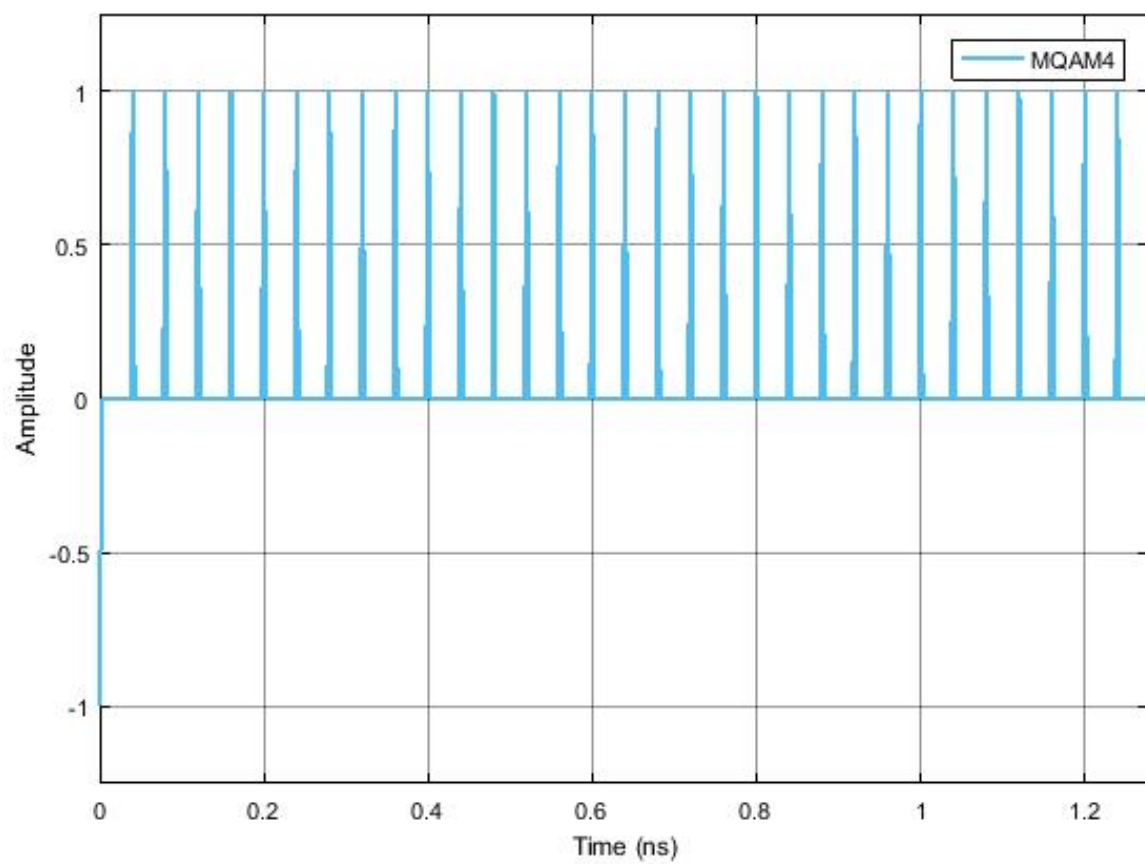


Figura 6.5: Example of the type of signal generated by this block for a binary sequence 0100...

## 6.6 Homodyne receiver

This block of code simulates the reception and demodulation of an optical signal (which is the input signal of the system) outputting a binary signal. A simplified schematic representation of this block is shown in figure 6.6.

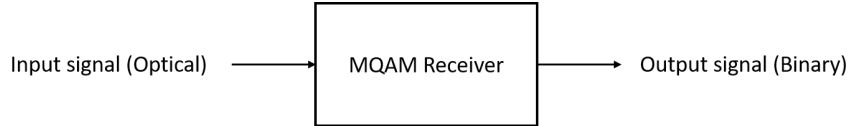


Figura 6.6: Basic configuration of the MQAM receiver

### Functional description

This block accepts one optical input signal and outputs one binary signal that corresponds to the M-QAM demodulation of the input signal. It is a complex block (as it can be seen from figure 6.7) of code made up of several simpler blocks whose description can be found in the *lib* repository.

It can also be seen from figure 6.7 that there's an extra internal (generated inside the homodyne receiver block) input signal generated by the *Clock*. This block is used to provide the sampling frequency to the *Sampler*.

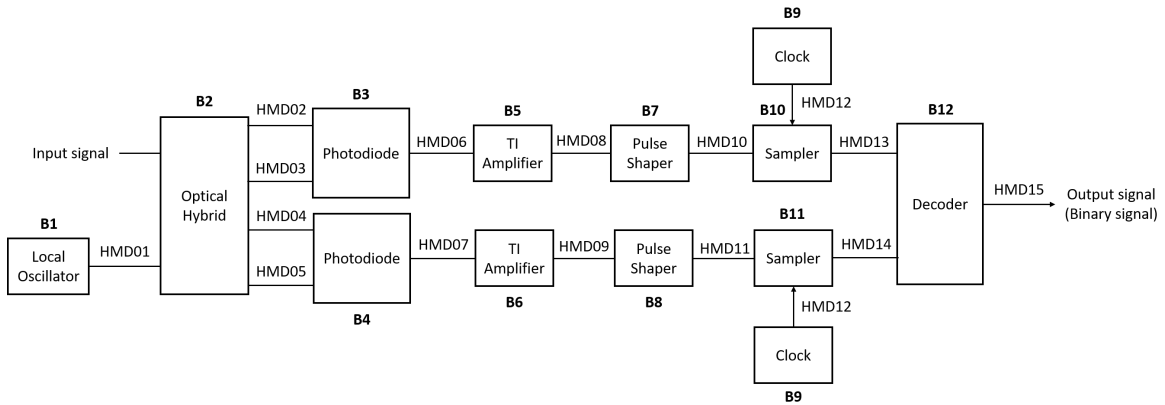


Figura 6.7: Schematic representation of the block homodyne receiver.

### Input parameters

This block has some input parameters that can be manipulated by the user in order to change the basic configuration of the receiver. Each parameter has associated a function that allows for its change. In the following table (table 6.2) the input parameters and corresponding functions are summarized.

Input parameters	Function	Type	Accepted values
IQ amplitudes	setIqAmplitudes	Vector of coordinate points in the I-Q plane	<b>Example</b> for a 4-qam mapping: { { 1.0, 1.0 }, { -1.0, 1.0 }, { -1.0, -1.0 }, { 1.0, -1.0 } }
Local oscillator power (in dBm)	setLocalOscillatorOpticalPower_dBm	double(t_real)	Any double greater than zero
Local oscillator phase	setLocalOscillatorPhase	double(t_real)	Any double greater than zero
Responsivity of the photodiodes	setResponsivity	double(t_real)	$\in [0,1]$
Amplification (of the TI amplifier)	setAmplification	double(t_real)	Positive real number
Noise amplitude (introduced by the TI amplifier)	setNoiseAmplitude	double(t_real)	Real number greater than zero
Samples to skip	setSamplesToSkip	int(t_integer)	
Save internal signals	setSaveInternalSignals	bool	True or False
Sampling period	setSamplingPeriod	double	Given by $symbolPeriod / samplesPerSymbol$

Tabela 6.1: List of input parameters of the block MQAM receiver

## Methods

HomodyneReceiver(vector<Signal \*> &inputSignal, vector<Signal \*> &outputSignal)  
(**constructor**)

void setIqAmplitudes(vector<t\_iqValues> iqAmplitudesValues)

vector<t\_iqValues> const getIqAmplitudes(void)

void setLocalOscillatorSamplingPeriod(double sPeriod)

void setLocalOscillatorOpticalPower(double opticalPower)

void setLocalOscillatorOpticalPower\_dBm(double opticalPower\_dBm)

void setLocalOscillatorPhase(double lOscillatorPhase)

void setLocalOscillatorOpticalWavelength(double lOscillatorWavelength)

void setSamplingPeriod(double sPeriod)

```
void setResponsivity(t_real Responsivity)

void setAmplification(t_real Amplification)

void setNoiseAmplitude(t_real NoiseAmplitude)

void setImpulseResponseTimeLength(int impResponseTimeLength)

void setFilterType(PulseShaperFilter fType)

void setRollOffFactor(double rOffFactor)

void setClockPeriod(double per)

void setSamplesToSkip(int sToSkip)
```



### **Input Signals**

**Number:** 1

**Type:** Optical signal

### **Output Signals**

**Number:** 1

**Type:** Binary signal

### **Example**

**Suggestions for future improvement**

## 6.7 IQ modulator

This block accepts one input signal continuous in both time and amplitude and it can produce either one or two output signals. It generates an optical signal and it can also generate a binary signal.

### Input Parameters

**Parameter:** outputOpticalPower{1e-3}  
(double)

**Parameter:** outputOpticalWavelength{1550e-9}  
(double)

**Parameter:** outputOpticalFrequency{speed\_of\_light/outputOpticalWavelength}  
(double)

### Methods

```
IqModulator(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig){};
```

```
void initialize(void);
```

```
bool runBlock(void);
```

```
void setOutputOpticalPower(double outOpticalPower)
```

```
void setOutputOpticalPower_dBm(double outOpticalPower_dBm)
```

```
void setOutputOpticalWavelength(double outOpticalWavelength)
```

```
void setOutputOpticalFrequency(double outOpticalFrequency)
```

### Functional Description

This block takes the two parts of the signal: in phase and in amplitude and it combines them to produce a complex signal that contains information about the amplitude and the phase.

This complex signal is multiplied by  $\frac{1}{2}\sqrt{\text{outputOpticalPower}}$  in order to reintroduce the information about the energy (or power) of the signal. This signal corresponds to an optical signal and it can be a scalar or have two polarizations along perpendicular axis. It is the signal that is transmitted to the receptor.

The binary signal is sent to the Bit Error Rate (BER) measurement block.

### Input Signals

**Number** : 2

**Type** : Sequence of impulses modulated by the filter (ContinuousTimeContinuousAmplitude))

### Output Signals

**Number** : 1 or 2

**Type** : Complex signal (optical) (ContinuousTimeContinuousAmplitude) and binary signal (DiscreteTimeDiscreteAmplitude)

### Example

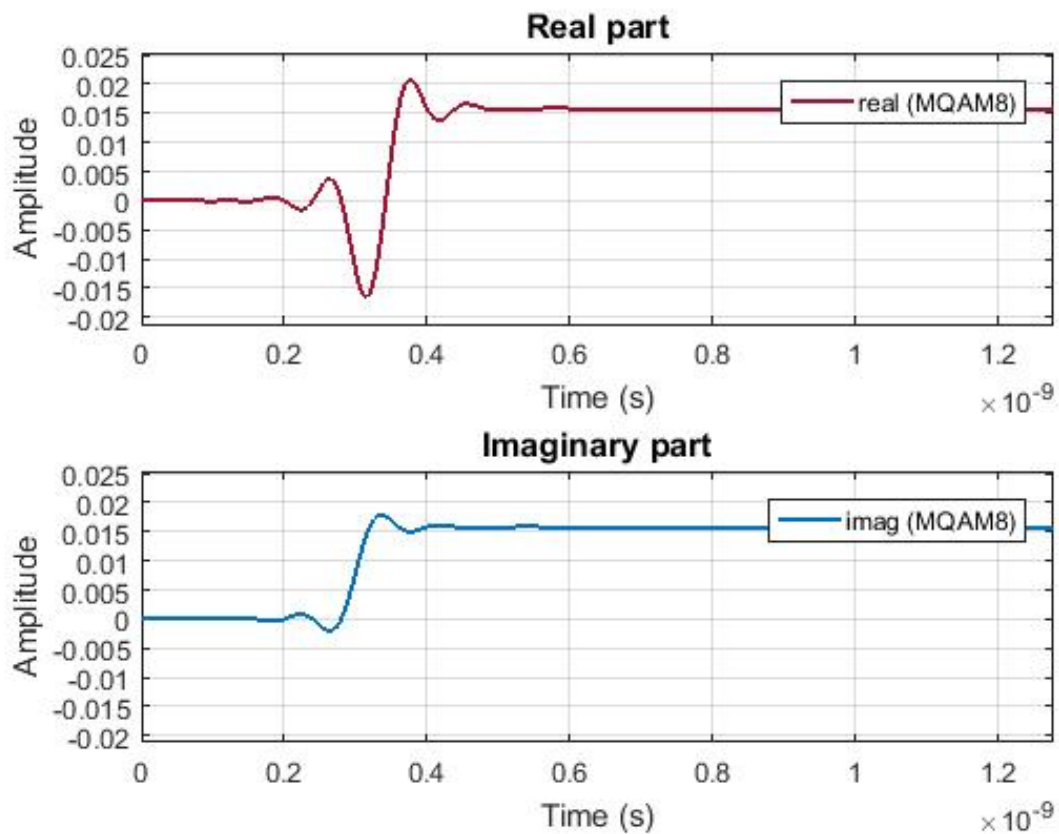


Figura 6.8: Example of a signal generated by this block for the initial binary signal 0100...

## 6.8 Local Oscillator

This block simulates a local oscillator which can have shot noise or not. It produces one output complex signal and it doesn't accept input signals.

### Input Parameters

**Parameter:** opticalPower{ 1e-3 }

**Parameter:** wavelength{ 1550e-9 }

**Parameter:** frequency{ SPEED\_OF\_LIGHT / wavelength }

**Parameter:** phase{ 0 }

**Parameter:** samplingPeriod{ 0.0 }

**Parameter:** shotNoise{ false }

### Methods

LocalOscillator()

```
LocalOscillator(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)
:Block(InputSig, OutputSig){};
```

```
void initialize(void);
```

```
bool runBlock(void);
```

```
void setSamplingPeriod(double sPeriod);
```

```
void setOpticalPower(double oPower);
```

```
void setOpticalPower_dBm(double oPower_dBm);
```

```
void setWavelength(double wlength);
```

```
void setPhase(double lOscillatorPhase);
```

```
void setShotNoise(bool sNoise);
```

### Functional description

This block generates a complex signal with a specified phase given by the input parameter *phase*.

It can have shot noise or not which corresponds to setting the *shotNoise* parameter to True or False, respectively. If there isn't shot noise the the output of this block is given by  $0.5 * \sqrt{\text{OpticalPower}} * \text{ComplexSignal}$ . If there's shot noise then a random gaussian distributed noise component is added to the *OpticalPower*.

**Input Signals**

**Number:** 0

**Output Signals**

**Number:** 1

**Type:** Optical signal

**Examples**

**Sugestions for future improvement**

## 6.9 MQAM mapper

This block does the mapping of the binary signal using a  $m$ -QAM modulation. It accepts one input signal of the binary type and it produces two output signals which are a sequence of 1's and -1's.

### Input Parameters

**Parameter:** `m{4}`  
(`m` should be of the form  $2^n$  with `n` integer)

**Parameter:** `iqAmplitudes{{ 1.0, 1.0 }, { -1.0, 1.0 }, { -1.0, -1.0 }, { 1.0, -1.0 }}`

### Methods

```
MQamMapper(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)
:Block(InputSig, OutputSig) {};
```

```
void initialize(void);
```

```
bool runBlock(void);
```

```
void setM(int mValue);
```

```
void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues);
```

### Functional Description

In the case of  $m=4$  this block attributes to each pair of bits a point in the I-Q space. The constellation used is defined by the *iqAmplitudes* vector. The constellation used in this case is illustrated in figure 6.9.

### Input Signals

**Number** : 1

**Type** : Binary (DiscreteTimeDiscreteAmplitude)

### Output Signals

**Number** : 2

**Type** : Sequence of 1's and -1's (DiscreteTimeDiscreteAmplitude)

### Example

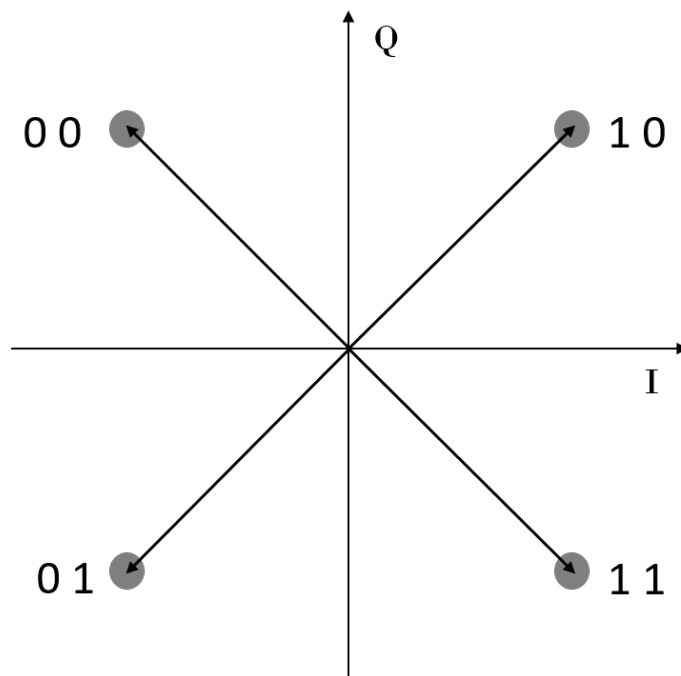


Figura 6.9: Constellation used to map the signal for  $m=4$

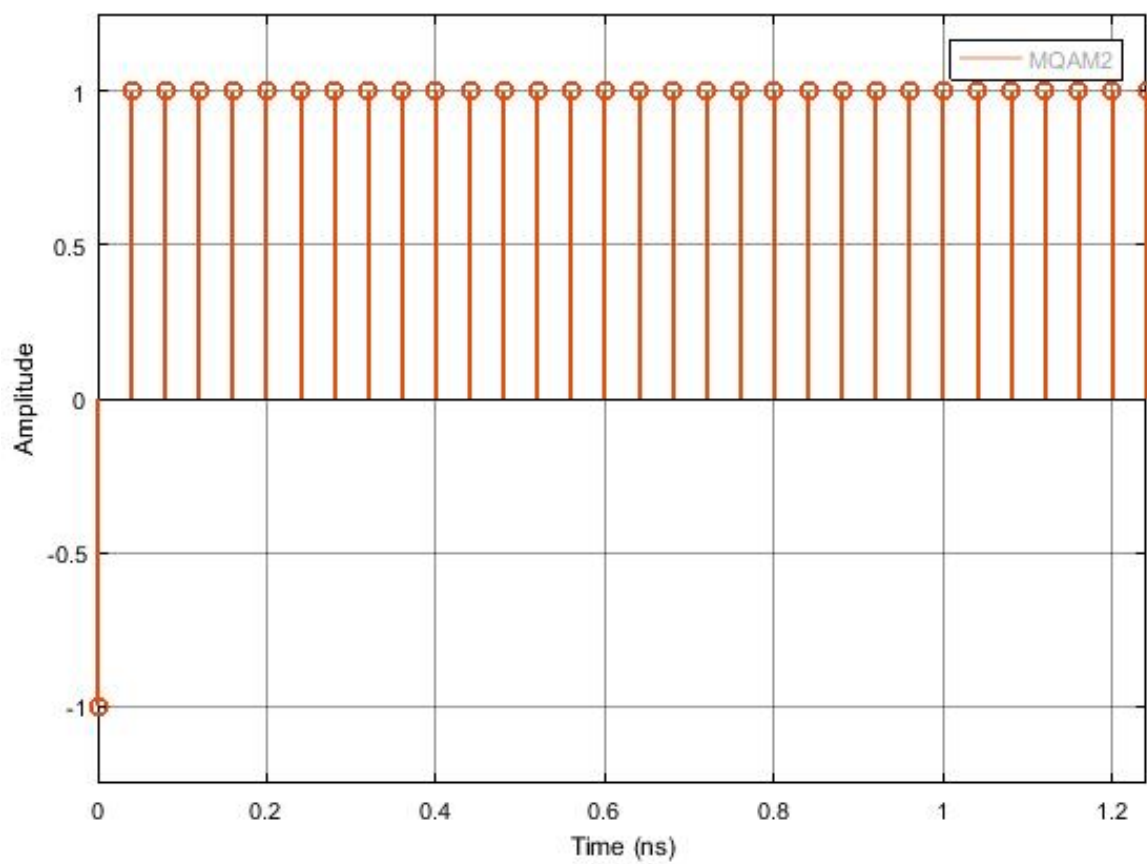


Figura 6.10: Example of the type of signal generated by this block for the initial binary signal 0100...



## 6.10 MQAM transmitter

This block generates a MQAM optical signal. It can also output the binary sequence. A schematic representation of this block is shown in figure 6.11.

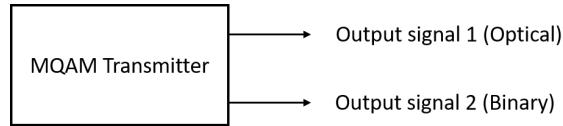


Figura 6.11: Basic configuration of the MQAM transmitter

### Functional description

This block generates an optical signal (output signal 1 in figure 6.12). The binary signal generated in the internal block Binary Source (block B1 in figure 6.12) can be used to perform a Bit Error Rate (BER) measurement and in that sense it works as an extra output signal (output signal 2 in figure 6.12).

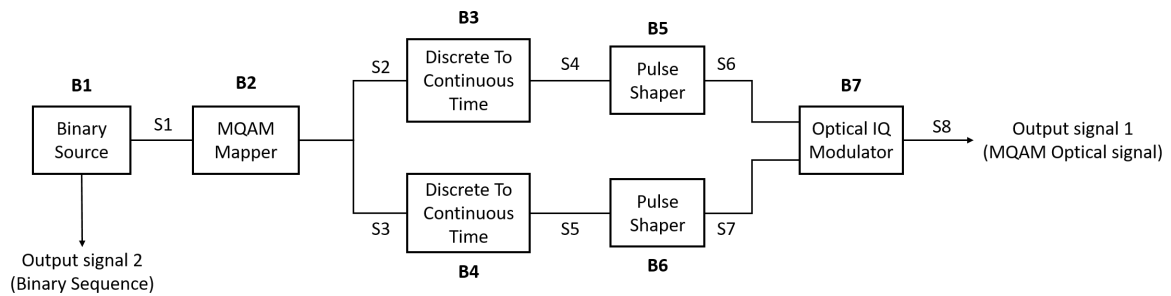


Figura 6.12: Schematic representation of the block MQAM transmitter.

### Input parameters

This block has a special set of functions that allow the user to change the basic configuration of the transmitter. The list of input parameters, functions used to change them and the values that each one can take are summarized in table 6.2.

Input parameters	Function	Type	Accepted values
Mode	setMode()	string	PseudoRandom Random DeterministicAppendZeros DeterministicCyclic
Number of bits generated	setNumberOfBits()	int	Any integer
Pattern length	setPatternLength()	int	Real number greater than zero
Number of bits	setNumberOfBits()	long	Integer number greater than zero
Number of samples per symbol	setNumberOfSamplesPerSymbol()	int	Integer number of the type $2^n$ with n also integer
Roll of factor	setRollOfFactor()	double	$\in [0,1]$
IQ amplitudes	setIqAmplitudes()	Vector of coordinate points in the I-Q plane	<b>Example</b> for a 4-qam mapping: { { 1.0, 1.0 }, { -1.0, 1.0 }, { -1.0, -1.0 }, { 1.0, -1.0 } }
Output optical power	setOutputOpticalPower()	int	Real number greater than zero
Save internal signals	setSaveInternalSignals()	bool	True or False

Tabela 6.2: List of input parameters of the block MQAM transmitter

## Methods

MQamTransmitter(vector<Signal \*> &inputSignal, vector<Signal \*> &outputSignal);  
(**constructor**)

void set(int opt);

void setMode(BinarySourceMode m)

BinarySourceMode const getMode(void)

void setProbabilityOfZero(double pZero)

double const getProbabilityOfZero(void)

void setBitStream(string bStream)

string const getBitStream(void)

```
void setNumberOfBits(long int nOfBits)

long int const getNumberOfBits(void)

void setPatternLength(int pLength)

int const getPatternLength(void)

void setBitPeriod(double bPeriod)

double const getBitPeriod(void)

void setM(int mValue) int const getM(void)

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)

vector<t_iqValues> const getIqAmplitudes(void)

void setNumberOfSamplesPerSymbol(int n)

int const getNumberOfSamplesPerSymbol(void)

void setRollOffFactor(double rOffFactor)

double const getRollOffFactor(void)

void setSeeBeginningOfImpulseResponse(bool sBeginningOfImpulseResponse)

double const getSeeBeginningOfImpulseResponse(void)

void setOutputOpticalPower(t_real outOpticalPower)

t_real const getOutputOpticalPower(void)

void setOutputOpticalPower_dBm(t_real outOpticalPower_dBm)

t_real const getOutputOpticalPower_dBm(void)
```

## Output Signals

**Number:** 1 optical and 1 binary (optional)

**Type:** Optical signal

## Example

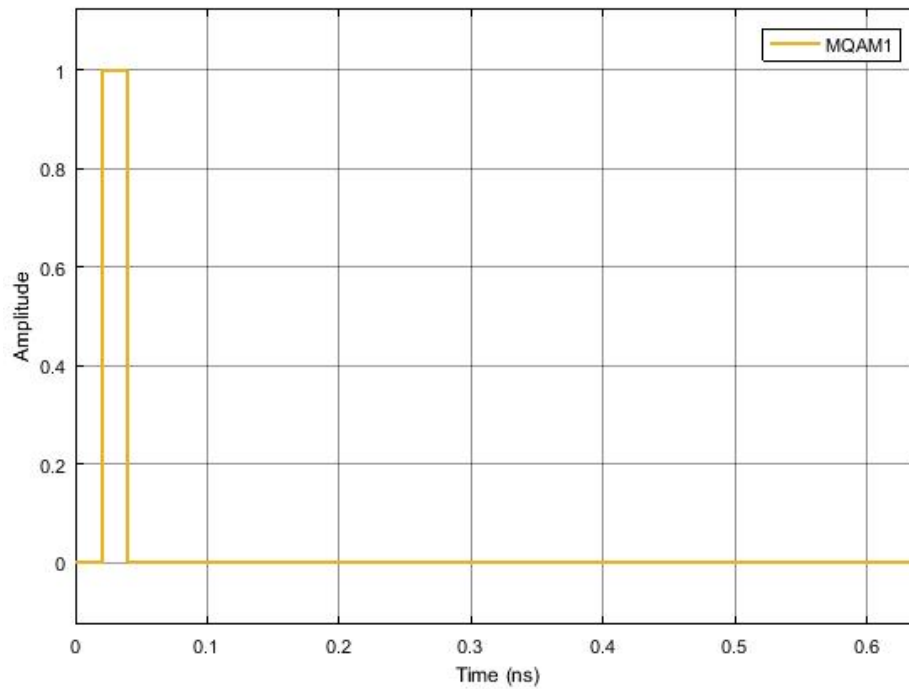


Figura 6.13: Example of the binary sequence generated by this block for a sequence 0100...

## Sugestions for future improvement

Add to the system another block similar to this one in order to generate two optical signals with perpendicular polarizations. This would allow to combine the two optical signals and generate an optical signal with any type of polarization.

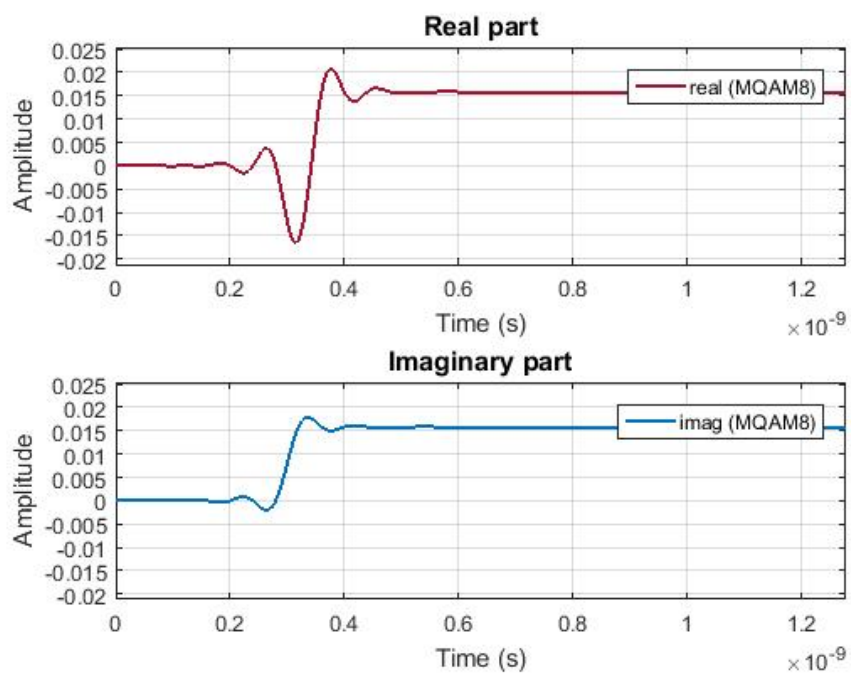


Figura 6.14: Example of the output optical signal generated by this block for a sequence 0100...

