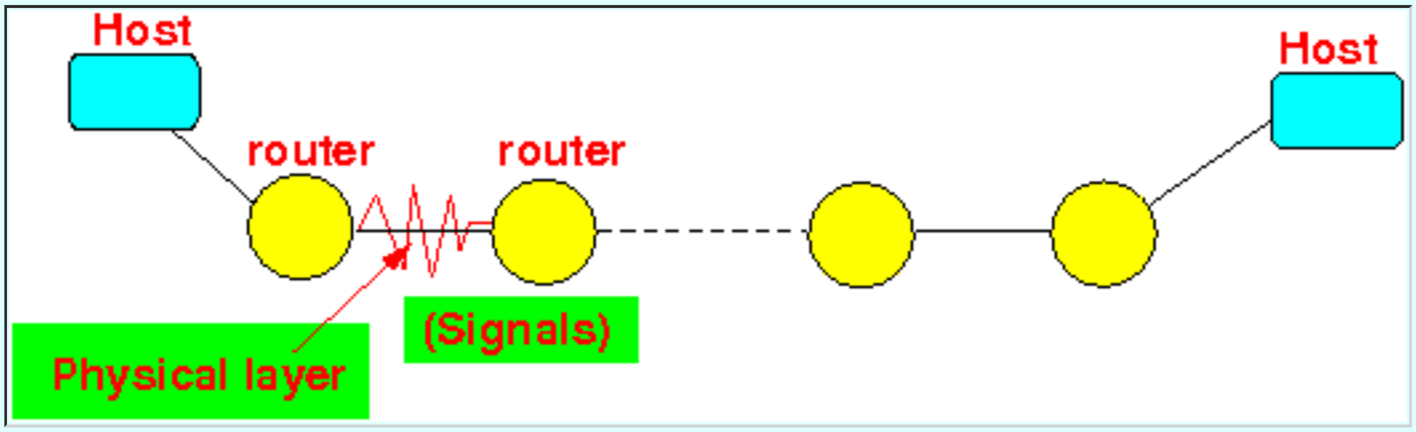
Physical Layer

* **The Physical Layer**
    * **Purpose** of the **Physical layer**:
        * The **Physical layer** is the **hardware *infra-structure*** used to ***convey* data** by ***transmitting* signals** from a **node** to one of its ***neighbor* nodes**:



    * **Important note:**
        * The **physical layer** deals ***only*** with **nodes** that can be reached ***directly***

Information, data and signals

- **Information and data**
  - **Data**:
    - **Data** = A **set** of ***qualitative*** or ***quantitative*** **values**
      - **E.g.: Dentist, Feb 2, 10:00 AM**
  - **Information:**
    - **Information** = the **meaning** that **humans** impart on **data**
      - **E.g.:** I have a **dentist appointment** on **Feb 2, at 10 AM**.
- ***Data* communication**
  - **Fact:**
    - We can ***only*** transmit **data**....
    - We **do *not*** transmit ***information***
- **Data and signals**
  - **Data transmission:**
    - In **computer communication**, **data** is **transmitted** using **(electrical) signals**
  - **Note:**
    - I **sometimes** use

- **Data**
- **Signal**

interchangeably, because:

- **Data** can be **stored** in the transmission **signals** !!!

- **Types of data**
  - There are **2 types** of **data**:
    - **Analog data**:
      - **Analog data varies** over a ***continous* range** of **values**

      Examples:

      - **Temperature** over a **period** of time
      - **Voice** (waves are varying continuously)

    - **Digital data**:
      - **Discrete data** takes on a ***finite* set** of **values**

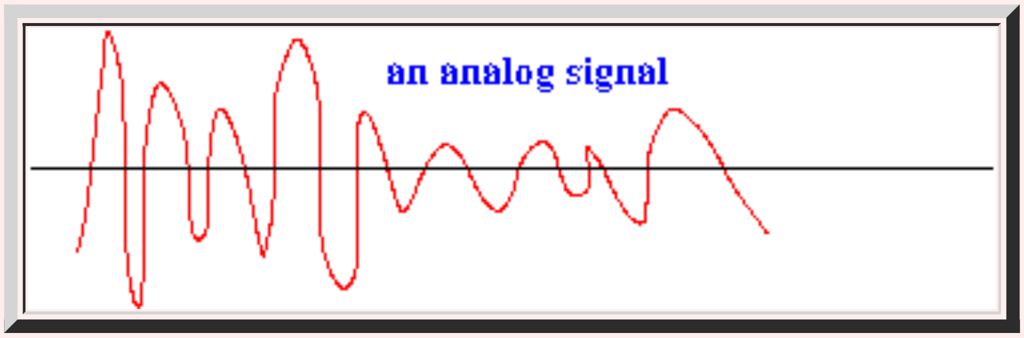
      Examples:

      - ***Current* temperature (rounded to 1 decimal)** (ranged from **-40.0 .. 140.0**)
      - **Marital status** (single, married, divorced, widowed)
      - **Character** (must be a **value** from an **alphabet**)
      - **Social Security Number** (between **000-00-0000** and **999-99-9999**)

- **Types of signaling methods (signals to send data)**
  - **Data communication** can use **2 different *signaling* methods**:
    - **Analog signaling**

- the **signals** in the **transmissions** can take on ***any* value** in some **(*continuous*) range** of **value**.

Example:

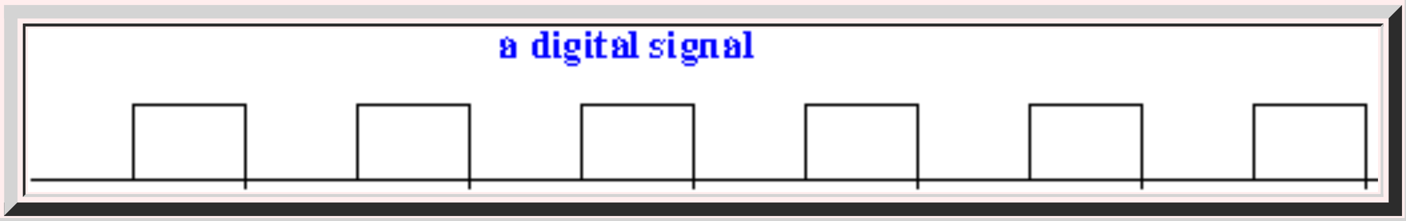


- **Digital signaling**

  - The **signals** in the **transmissions** cam ***only*** takes on a ***finite number*** **values**

    - **Commonly used**: **2 levels** (***binary* signal**)

Example: a ***binary* signal** (can ***only*** take on **2 different values**)



- **Transmitting analog/digital *data* using analog/digital *signals***
  - You can:
    - Transmit ***analog* information (data)** using ***analog* signals**
    - Transmit ***analog* information (data)** using ***digital* signals**
    - Transmit ***digital* information (data)** using ***analog* signals**
    - Transmit ***digital* information (data)** using ***digital* signals**

Transmitting *analog* data using *analog* signals

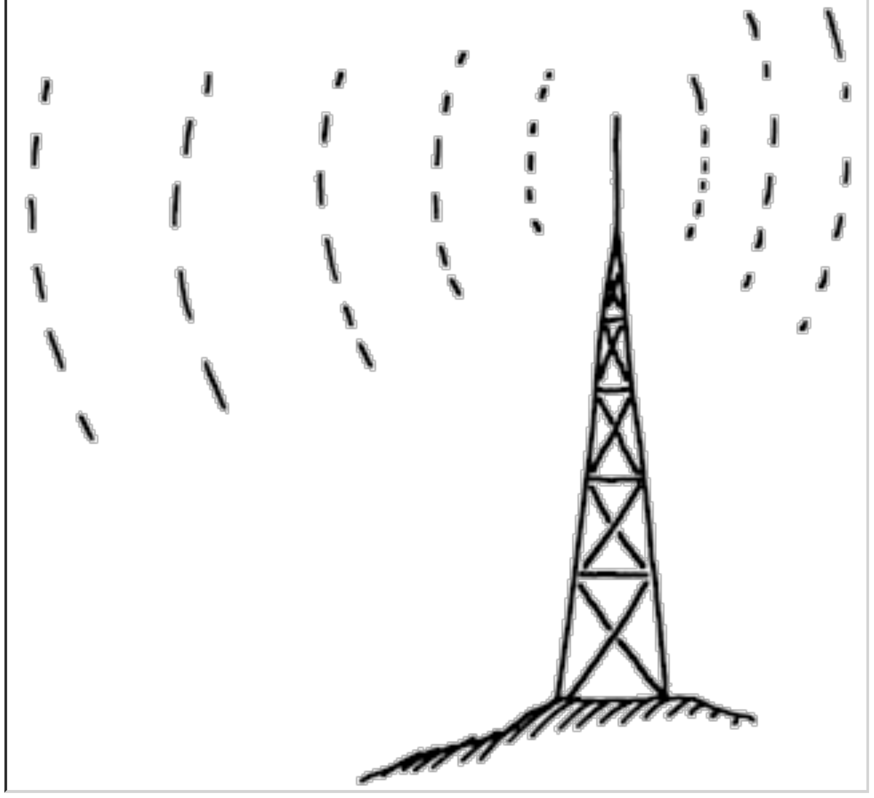
- **Analog data transmitted with *analog* signals**
  - **2 ways** to **transmit *analog* data** using ***analog* signals**:
    - **If** the **transmission medium** is **suitable** (i.e., it can ***carry* the transmitted signals**), then:
      - **transmit the signal**...

    - **If** the **transmission medium** is ***not* suitable** (i.e., it **does *not* *carry* the transmitted signals**), then:
      - ***Modulate* (super-impose)** the **signal** on a ***carrier* signal**
      - **Transmit** the ***modulated* signal**

- **Example a *naive* transmission**
  - **"Talking":**
    - The **atmosphere** is **suitable** to **carry audio waves**....

    - **Sound waves** can be **transmitted *naively***:

- **Intro to Modulation**
  - **Problem:**
    - **How** can **astronauts** **talk** on the **moon** ???
      - The **moon** does **not** have an **atmosphere**
      - **Sound wave** is ***not*** **suitable**
  - **Solution:**
    - ***Radio*** **waves** can **propagate** through **vacuum**:

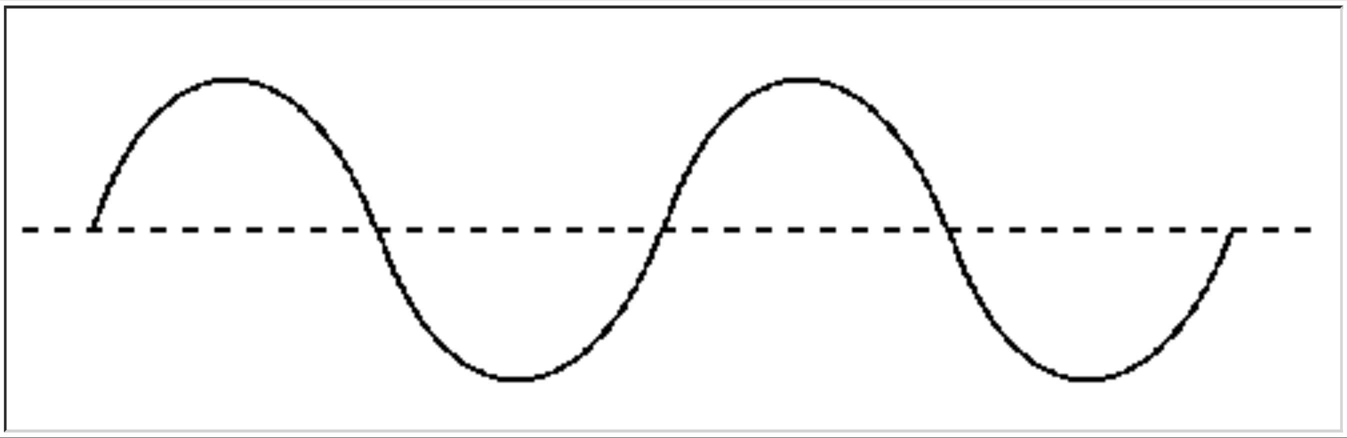


- We ***modulate*** the ***radio*** **wave** ***using*** a ***sound*** **wave**

- **Then transmit** the ***modulated*** **radio wave** (through **vacuum**)....

- ***What*** **is "Modulation"**
  - **Carrier wave:**
    - **Carrier wave** = a **sine wave** of a specific **frequence** and **amplitude** that **propagates** ***easily*** through the **transmission medium**

      Example: carrier wave

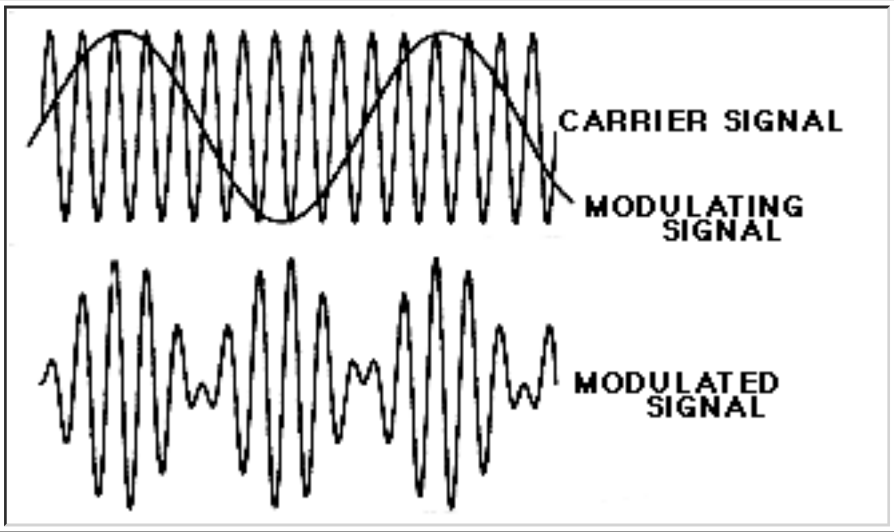
      

  - **Modulation:**
    - **Modulation** = ***change*** **a carrier wave using some (input) signal**

- *What* can you **change** in the **carrier wave**:
  - The **amplitude** of the **carrier wave** (= **Amplitude Modulation (AM)**)
  - The **frequency** of the **carrier wave** (= **Frequency Modulation (FM)**)
  - The **phase** of the **carrier wave** (= **Phase Modulation (PM)**)

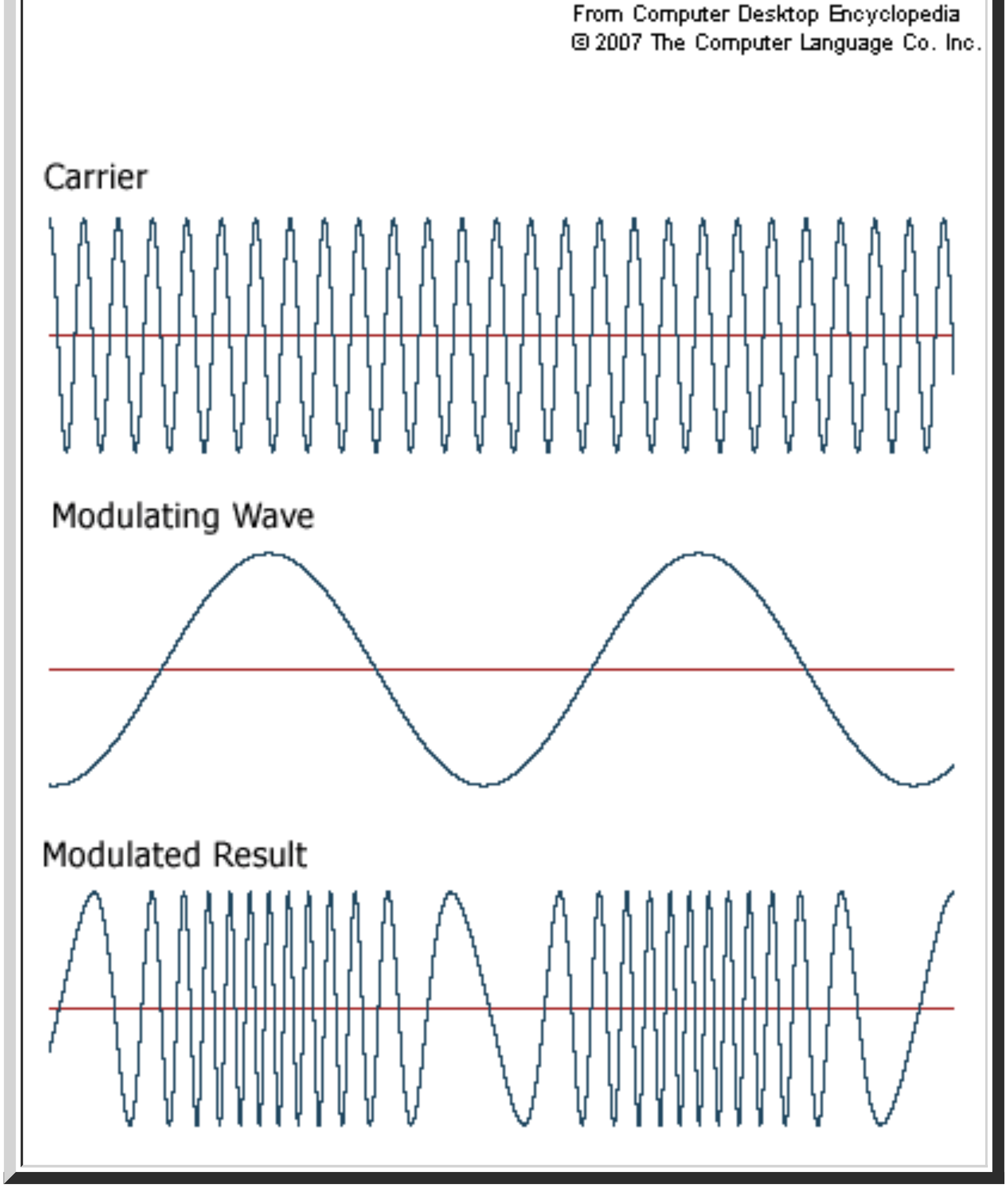
- **Amplitude Modulation**
  - ***Amplitude* modulation (AM)**):
    - the **amplitude** of the **carrier wave** is **changed** according to the **input signal**

  Example:

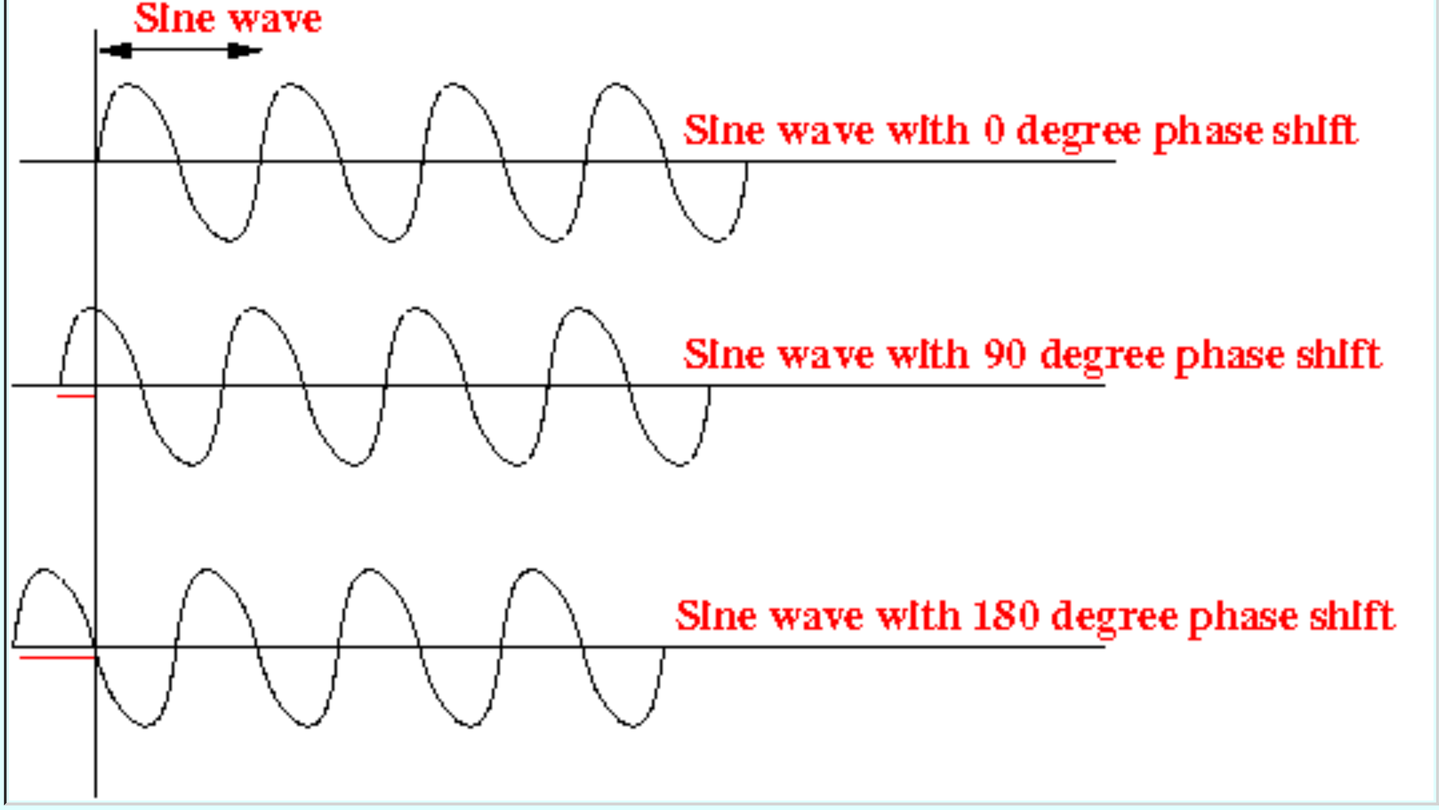


- **Frequency Modulation (FM)**
  - ***Frequency* modulation (FM)**):
    - the **frequency** of the **carrier wave** is **changed** according to the **input signal**

  Example:



- **Phase Modulation (PM)**
  - ***Phase* modulation (PM)**):
    - the **phase** of the **carrier wave** is **changed** according to the **input signal**
  - The **phase** of a **wave**:
    - The **phase** of a **sine wave** is the ***amount* of shift**:



- In **PM**, you **change** the ***instantaneous* phase** of the **carrier signal**.

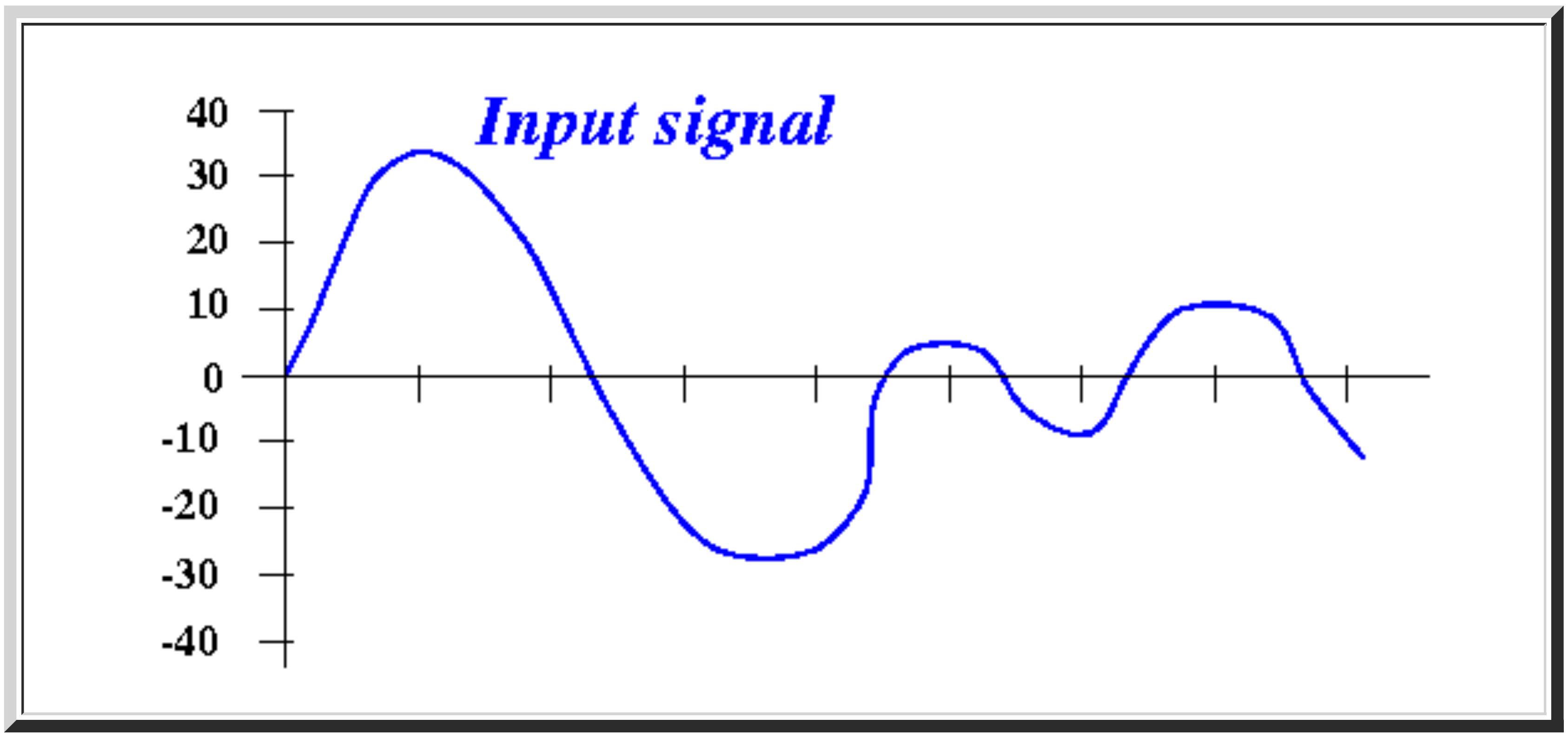
  - This is **very hard** to **detect**, even for **electronics** - let alone **humans** !!!

I will **not try** to **draw** a **continuous phase shifted** signal.....

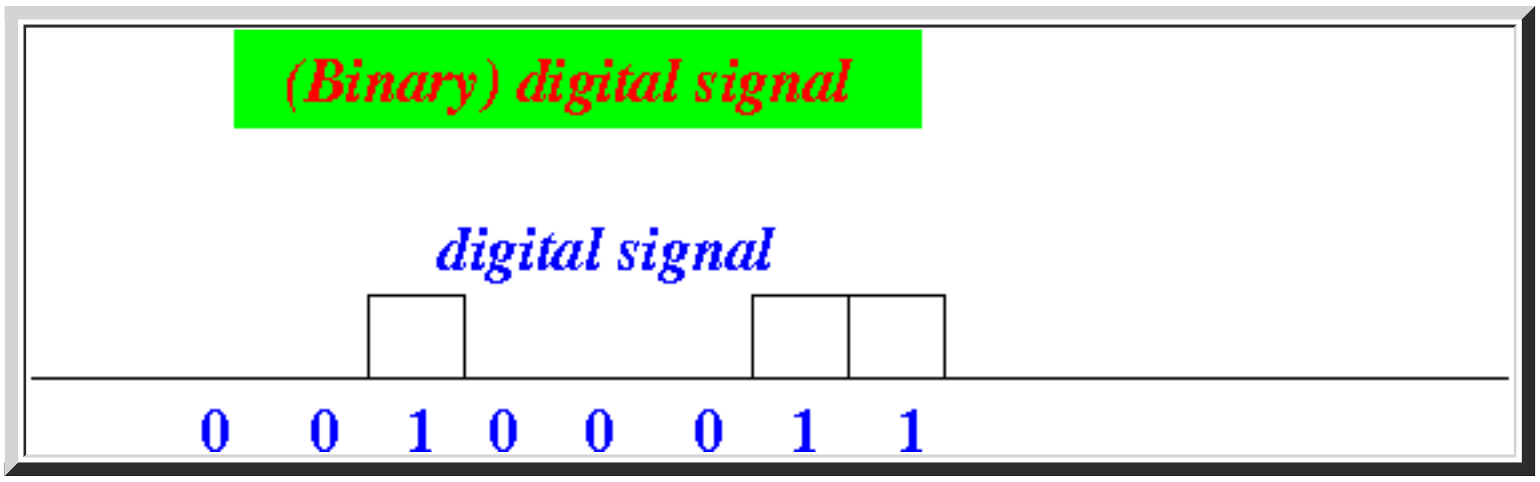
It's ***too* hard** :)

Converting *analog* data (signals) to *digital* data (signals)

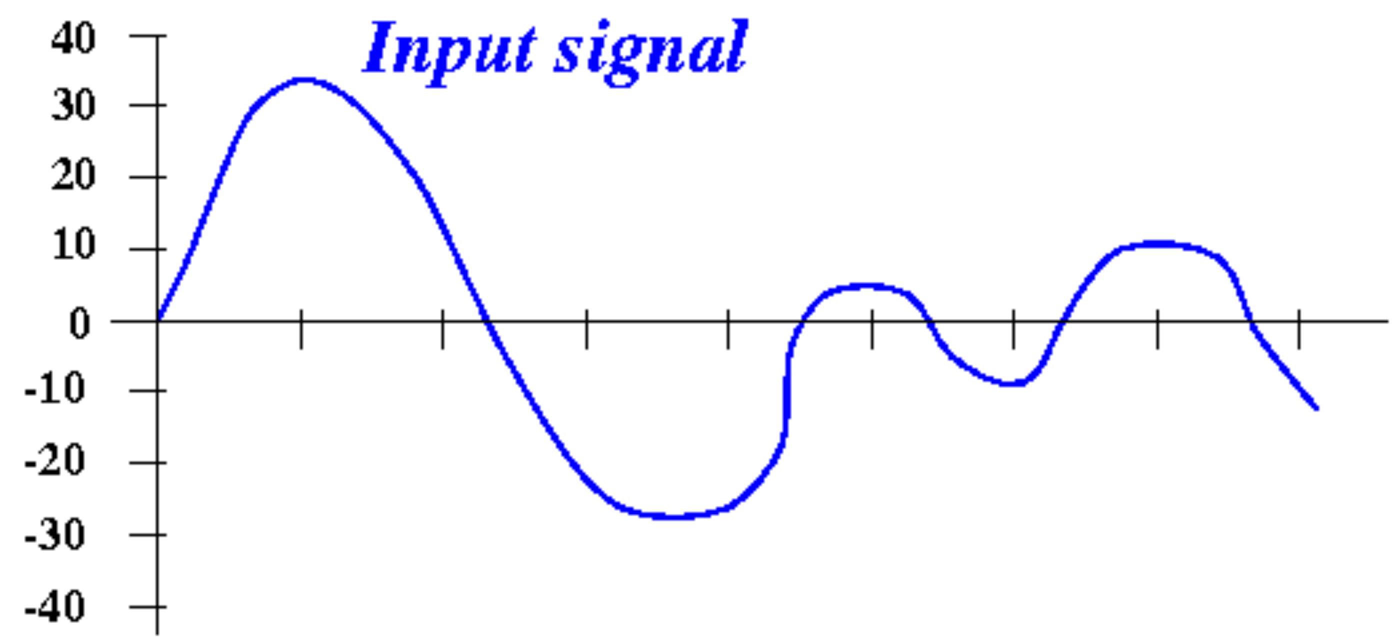
- **Transmitting analog data using digital signals**
  - **Analog** (input) **data**:



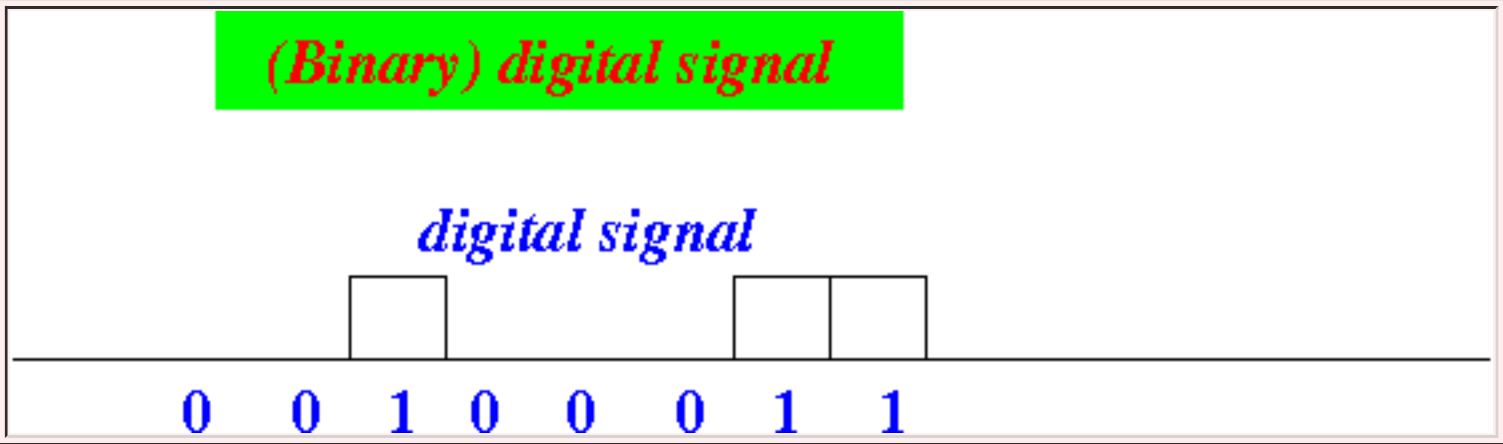
  - **Digital signals**:



  - **Problem Description:**
    - **How** do we **convert *this***:

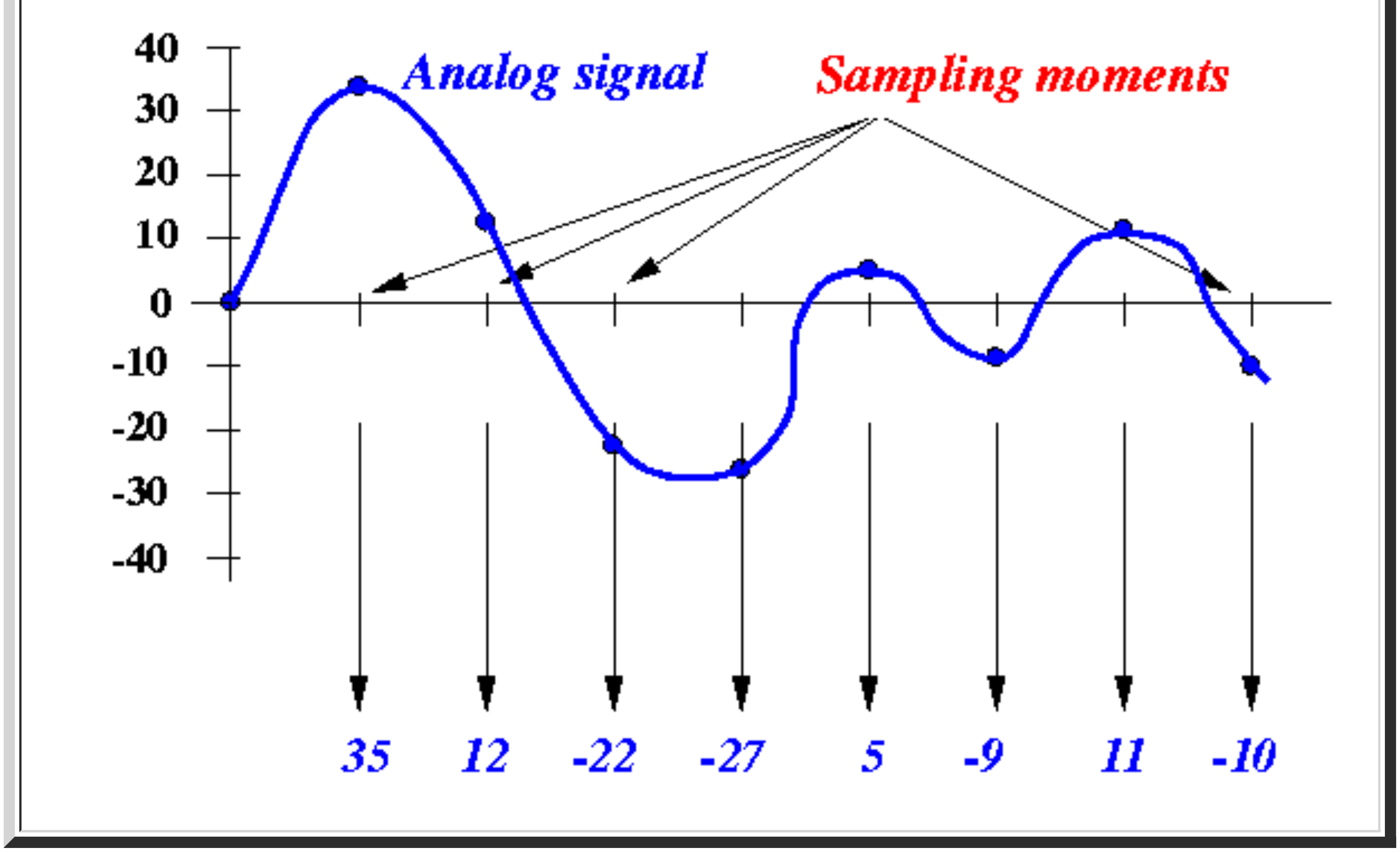


into ***this***:



- **Sampling**
  - **Sampling**:
    - **Measure** (and **record**) a **continuous signal** at ***regular*** **intervals**

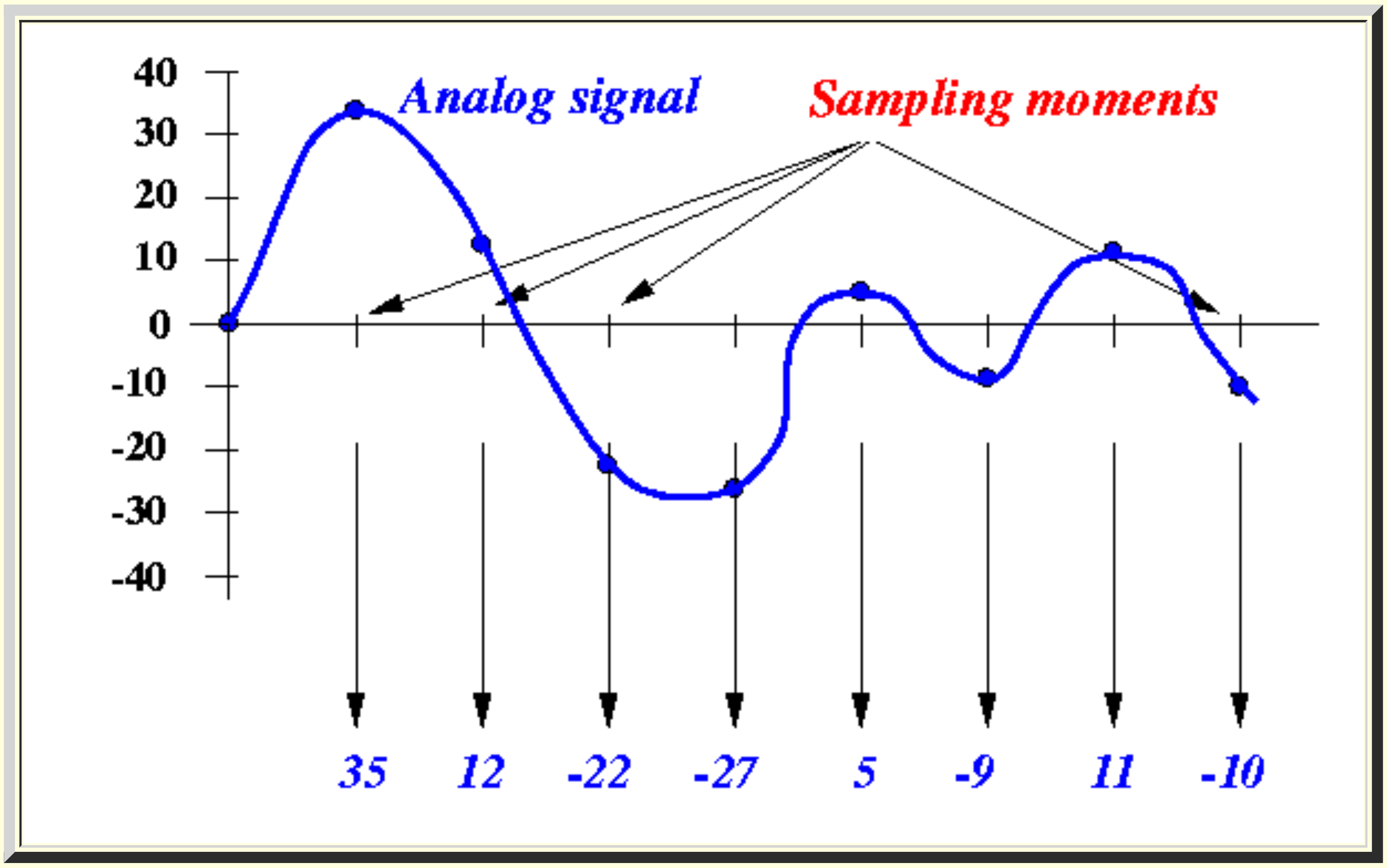
    Example:



- **Converting analog data (signal) to digital data (signal)**
  - **How to** transmit ***analog*** **data** using ***discrete*** **signals**

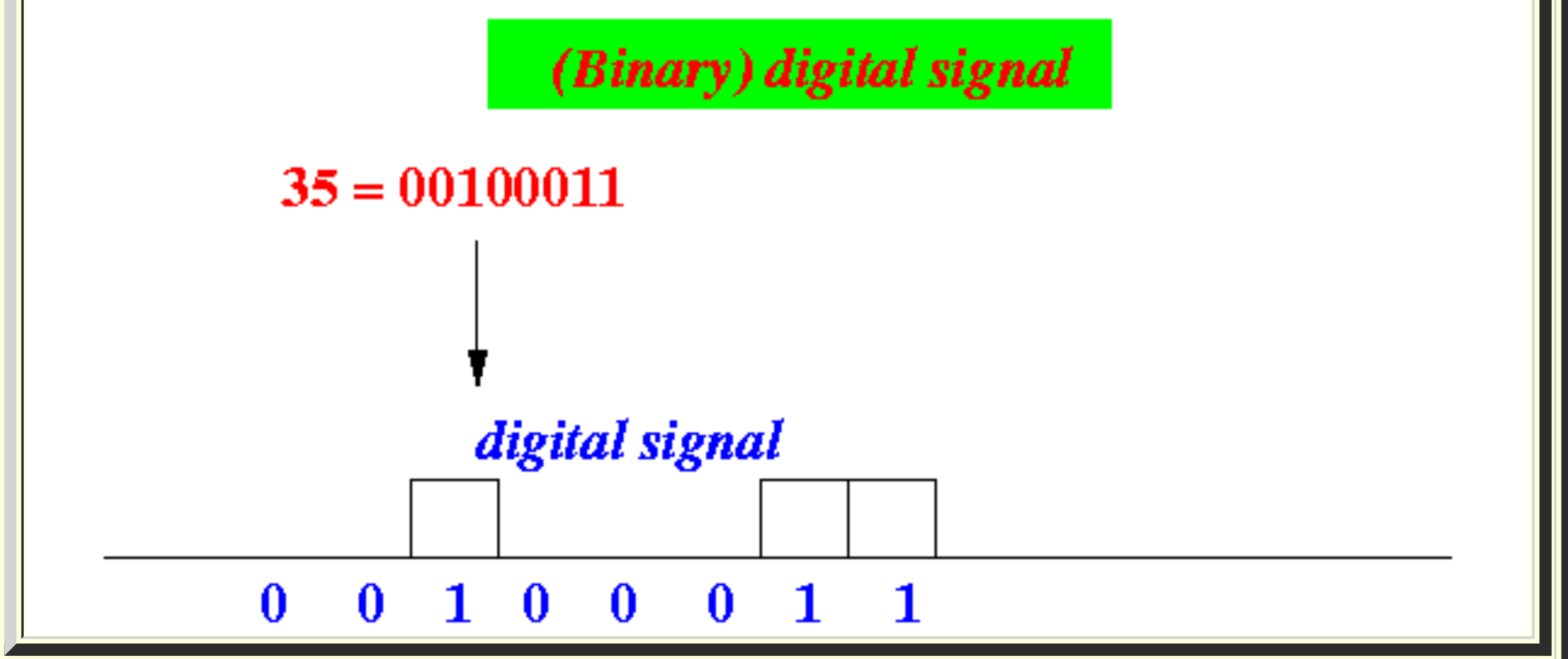
1. The **analog data** is ***first sampled*** into **serie** of ***numerical*** **values**

   Example:

   ---

2. **Each** of the ***numerical*** **value** is **encoded** into a **binary number** and transmitted using **(binary) digital signals**:



- A/D converters
  - **A/D converter (chip)**:
    - **A/D converter** = the **device (chip)** that **converts (samples) analog signals** to **digital signals**
      - **Wikipedia page:** click here

    You can **specify** a **sampling *rate***:

    - The **more *frequent*** you **sample** the input signal, the **better *accurate*** the **approximation** !!!

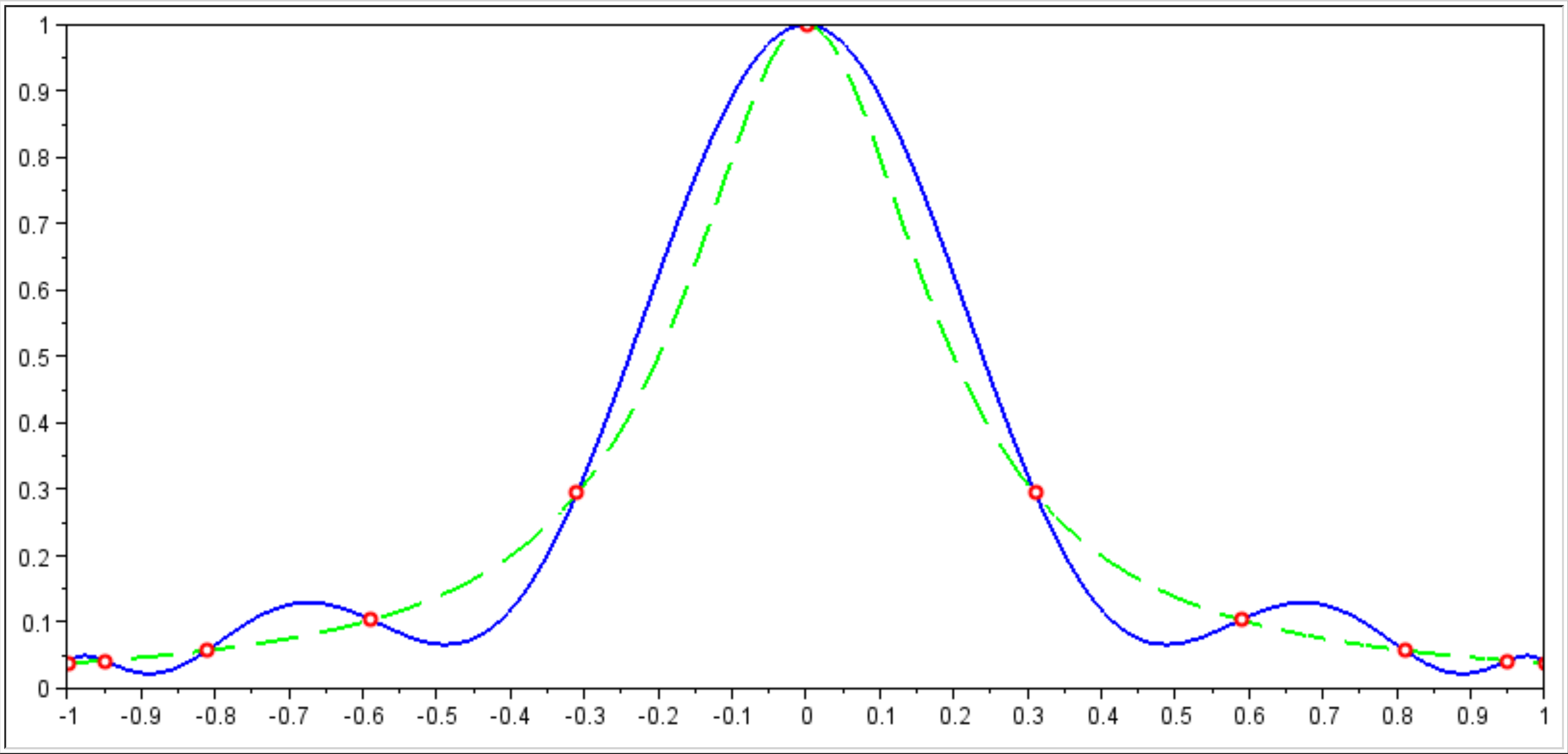
  - **Caveat:**
    - If you **sample** below a **certain threshold**, you **may not be able** to **reconstruct** the **original signal** using the **sampled values** !!!

    First, let's look at **how** you **re-construct** the **original signal** from the **samples**

Re-constructing the *analog* data (signals)

- **Re-constructing the *analog* data from the samples: *Lagrange inter-polation***
  - **Fact:** (from **Mathematics**)
    - You can **construct** a ***unique*** **polynomial function** of degree **n−1** that passes through **exactly** **n** points

      Examples:

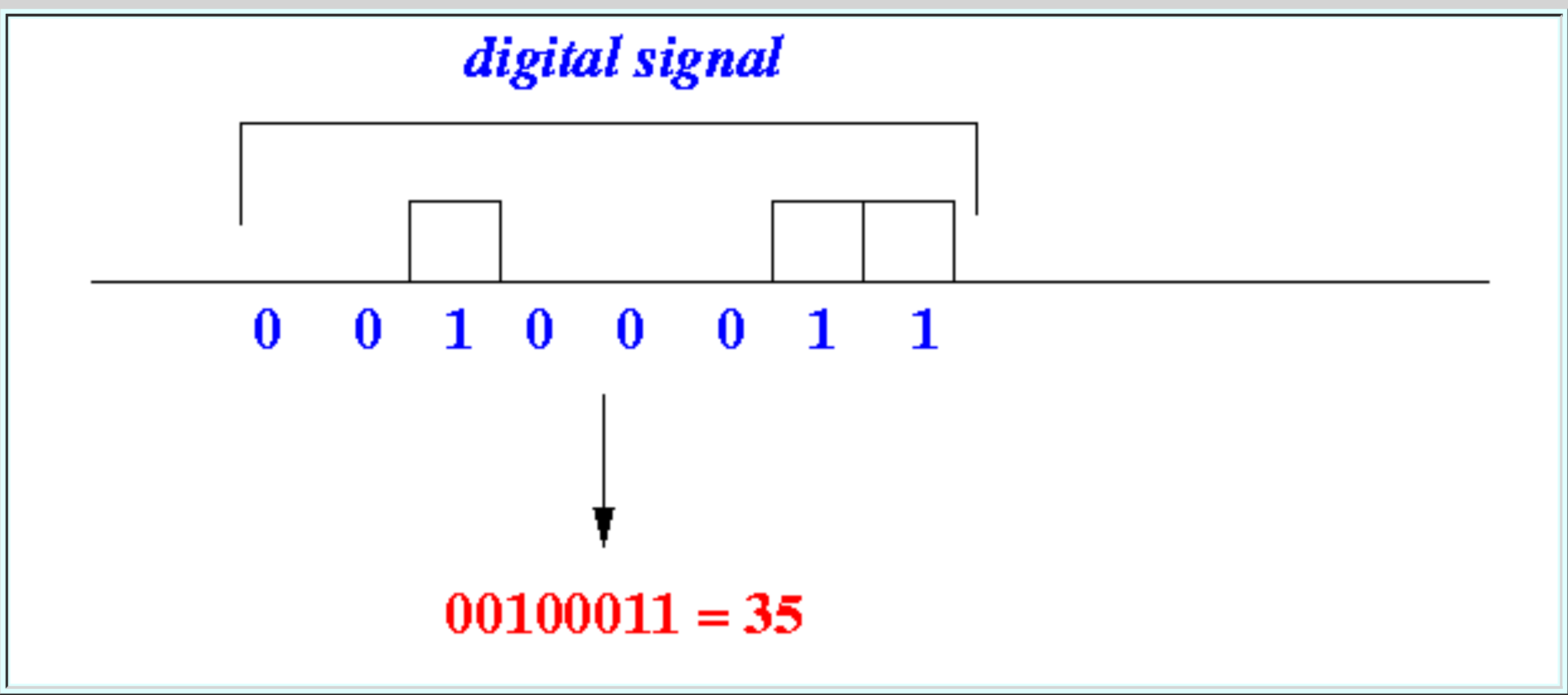
      

      The **dotted** plot is the **original signal**

  - **Langrange Polynomial:**
    - **Langrange Polynomial** = the **polynomial function** (of degree **n−1**) that you can **construct** when given **exactly** **n** data points

      See Wikipedia for more **details**: **click here**

  - **Re-constructing** the **analog data** from the **digital signal**:
    1. The **digital signals (pulses)** is **first** converted **back** to **digital values**:

    2. A **Interpolation Polynomial** is **computed** using ***n*** **data points** at a time

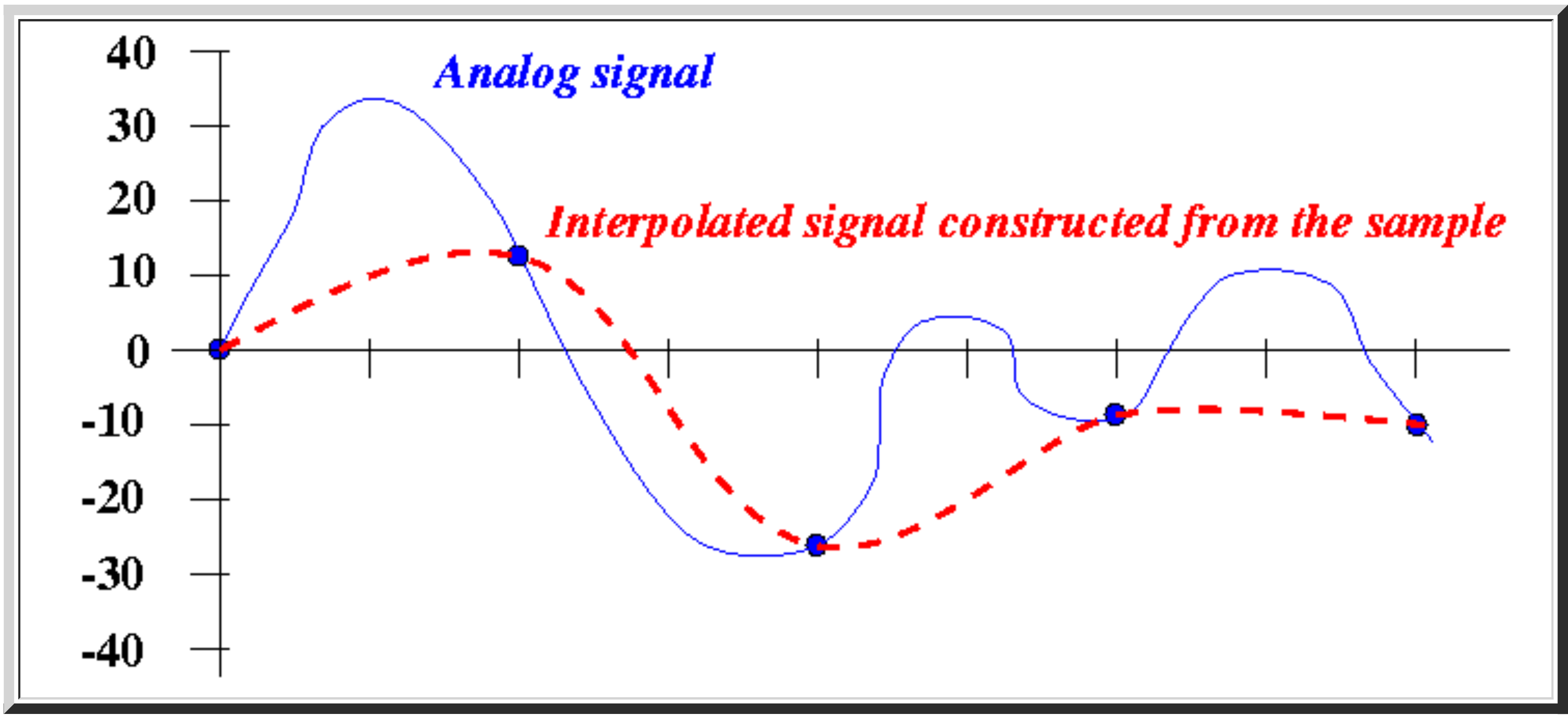
  - **Try it out**:

- Here's a **nice** **Language Polynomial** applet that you can **play with**: **click here**
- I **downloaded** a less nicer one: **click here**

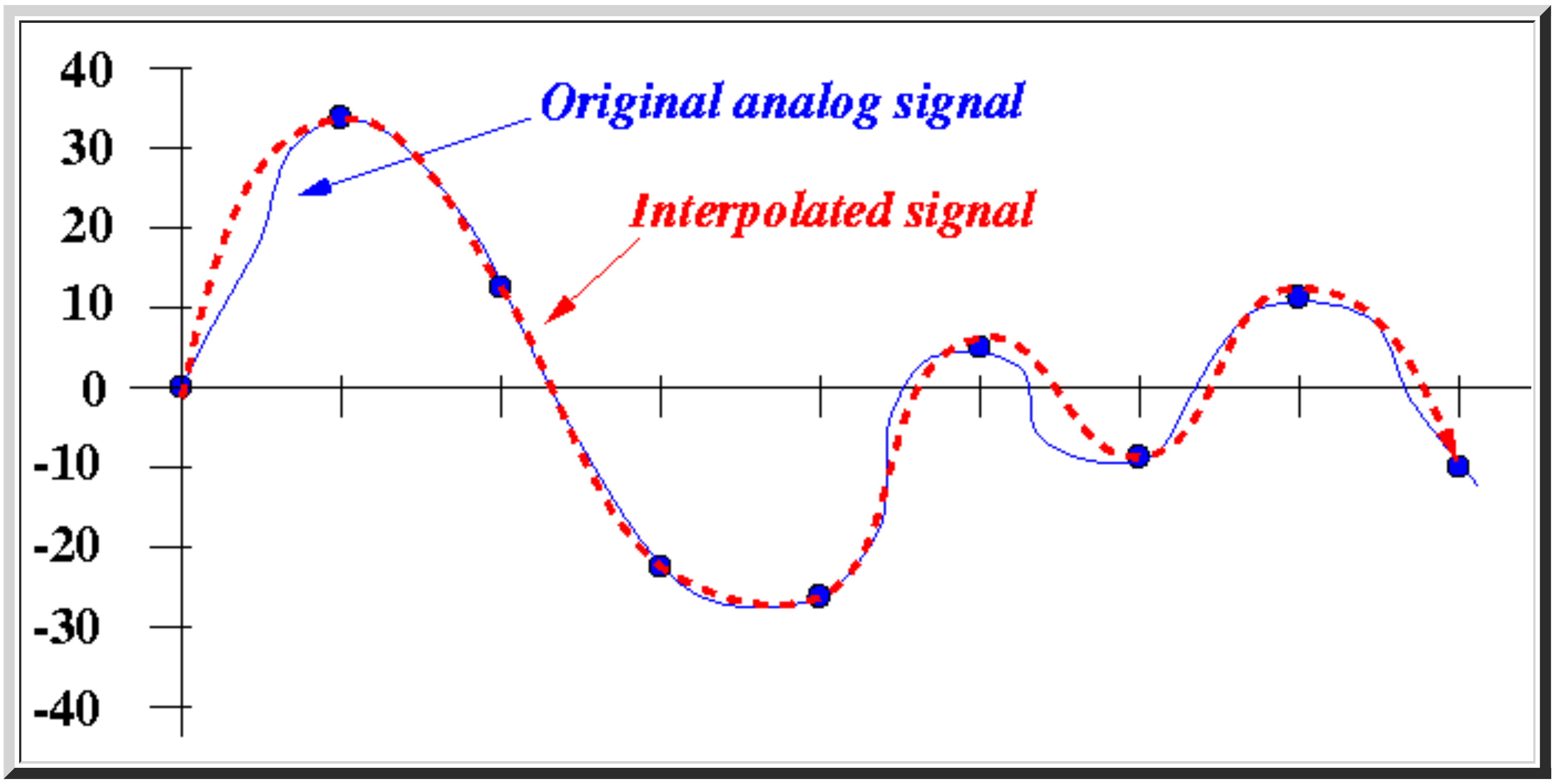
  Check out the **/home/cs455000/demo/Langrange** directory

Sampling rate and *accuracy* --- the Nyquist rate

- **Sampling rate and accuracy of the reconstructed analog signal**
  - **Re-constructed signal** and the **original signal** using a ***low* sampling rate**:



  - **Re-constructed signal** and the **original signal** using a ***higher* sampling rate**:



  - **Conclussion:**
    - The ***faster* (more frequent)** you **sample**, the ***more* accurate** the **approximation**
  - **$64,000 question:**

- **Should** we **sample** as **fast as possible** ????

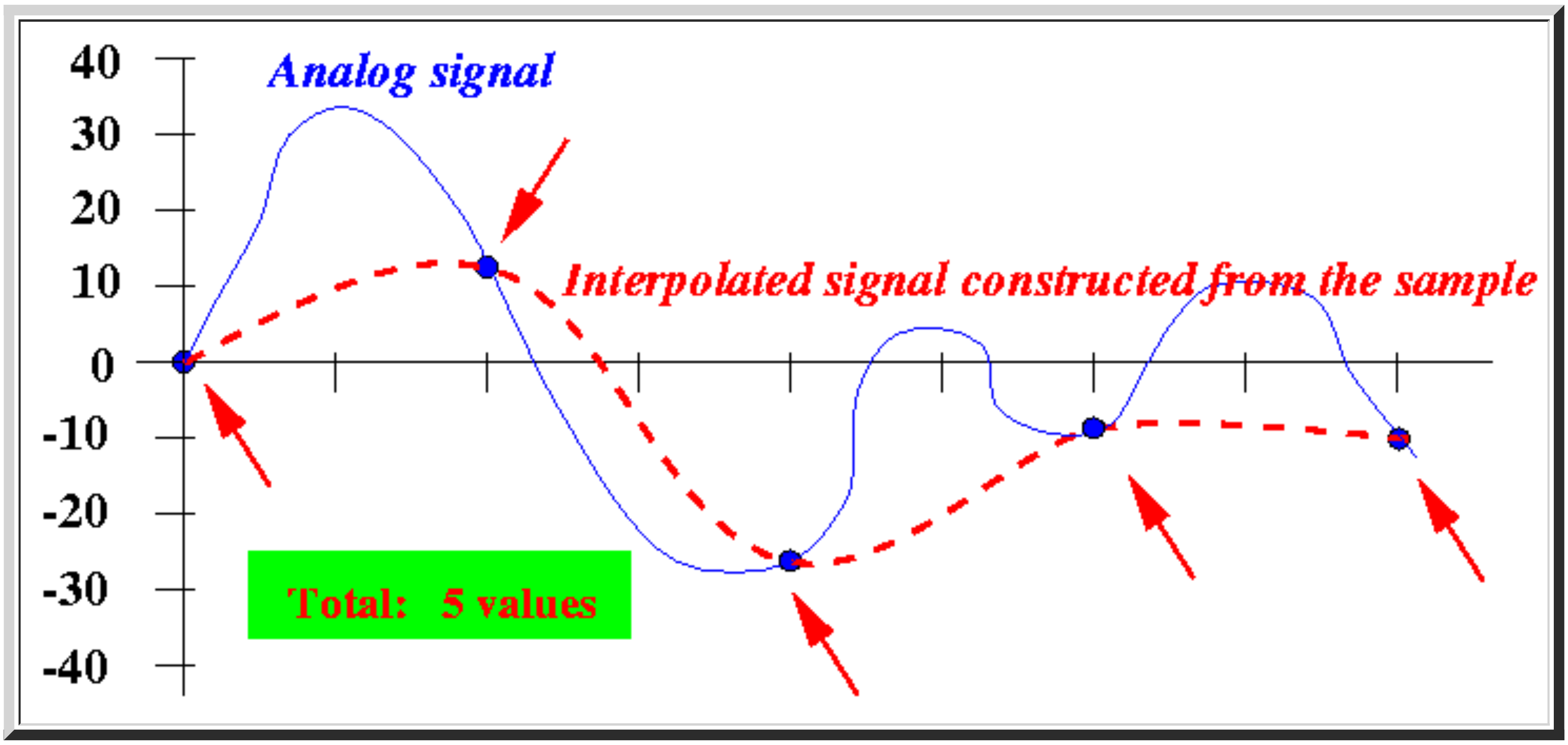
No !!!! The **reason** is **cost** !!

- **Transmission rate**
  - **Transmission rate**:
    - **Transmission rate** = the **amount** of **data (in bits)** transmitted per **time unit (sec)**

  - **Fact:**
    - We should **try** to **keep** the **transmission rate** as **low** as **possible**

      (**More data** sent = ***higher* cost**)

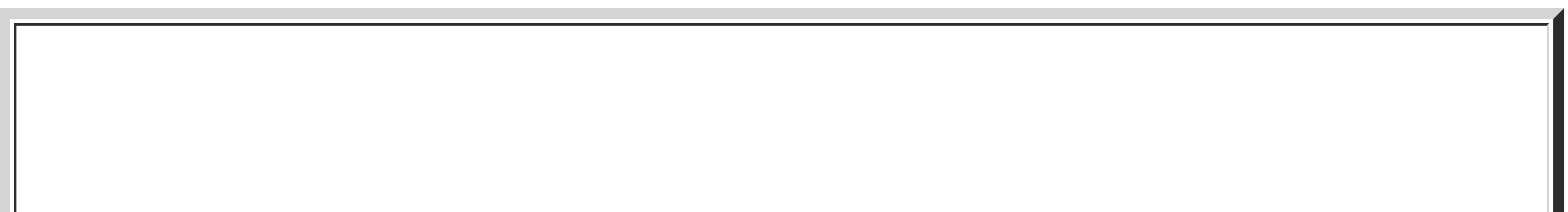
- **Sampling rate and transmission rate**
  - **Data rate** at a ***low* sampling rate**:

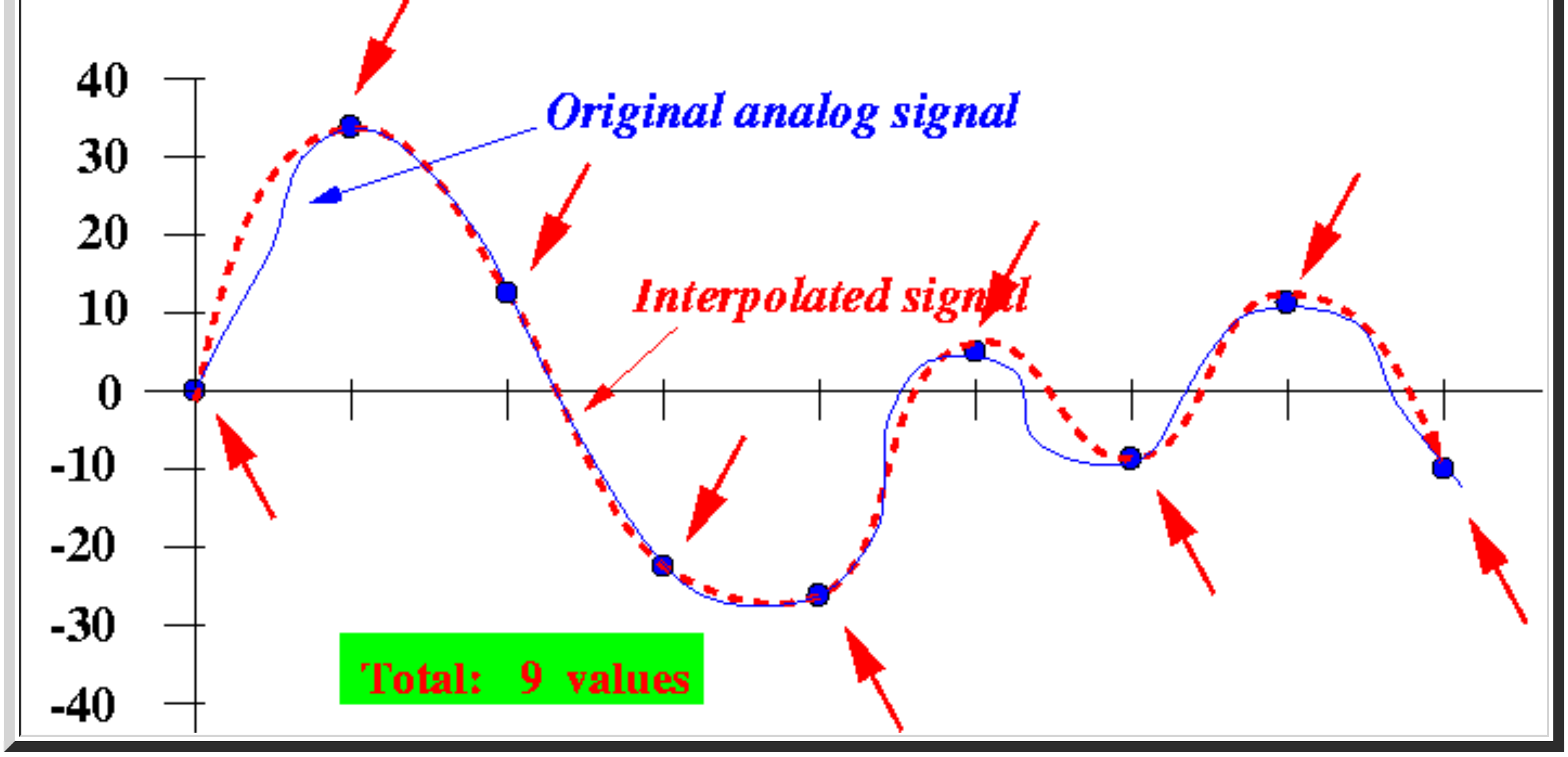


Conclusion:

- We **only** need to sent **5 values** when we **sample** at the **lower sampling rate**

  - **Data rate** at a ***higher* sampling rate**:





Conclusion:

- We must sent **9 values** when we **sample** at the **higher sampling rate** !!!!

- **Therefore:**
  - The ***faster*** **(more frequent)** you **sample**, the ***higher*** the **(transmission) data rate**

- **What is the best sampling rate ???**
  - **High sampling rate:**
    - **Good accuracy**
    - ***High* transmission data rate (costly)**
  - **Low sampling rate:**
    - ***Poor* accuracy**
    - **Low transmission data rate (cost-efficient)**
  - **Goal:** (what we want)
    - At ***which* data rate** can we achieve **good *enough* accuracy** ????

- **The Nyquist Sampling Rate**
  - The **Nyquist (sampling) rate:**

- **Nyquist (sampling) rate** = the ***minimum*** **sampling rate** required to **avoid aliasing** when **sampling a *continuous* signal**.

  See: Wikipedia

- **Aliasing**:

  - an **effect** that causes **different signals** to become ***indistinguishable*** when **sampled**.

  See: Wikipedia

- In **plain English**:

  - **Nyquist rate** = the ***lowest possible*** **sampling rate** that **permits** an ***accurate*** **reconstruction** of a **input signal** using **samples**

- **Disclaimer:**

  - We will **not** **derive** the **Nyquist rate** :-)...

    because it requires a **lot** of **background knowledge** and is **beyond the scope** of this course...

- I will **only** **state** the **result** from **Signal Processing**:

```
 Nyquist Rate = 2 × B

where:

   B = highest frequency found in the (analog) input signal
```

Application of the Nyquist Rate: digital CD quality audio

- **Nyquist rate to obtain digital CD quality audio**
  - **Problem description:**
    - We want to **store music** (which is ***analog*** **data**) as a **computer file** (which is ***digital*** **data**)
  - **Question:**
    - How **fast** do you need to **sample** to reconstruct **audible audio waves completely accurately** ?
  - **Pre-requisite**: a **fact** from **Biology**
    - **Human ear** can **hear** **sound waves** with **frequency** **≤ 20,000 Hz**
  - The **Nyquist sampling rate** for **human** **audible signal**:

```
Nyquist Sampling Rate = 2 × MaxFrequency    (MaxFreq = 20,000 Hz)
                      = 2 × 20000 Hz
                      = 40000 Hz

  (Meaning: 40000 samples per second)
```

Note:

```
1 Hz (Hertz) = 1 event (sample) per second
```

- **Sampling rate used by the music industry**
  - **Digital music:**
    - **Digital music** = **audio (music)** stored in **digital format**
    - Use **sampling** to **convert** **audio** to **digital** !!!
  - **Sampling rate** used by the **music industry**:
    - The **sample rate** used to obtain ***CD quality*** **audio** is: **44100 Hz**

(This rate is **higher** than the ***Nyquist* rate** of **40,000 Hz**)

- The **reason** is **historical**:

  - The **sampling rate** of **44.1 kHz** was **inherited** from **recording method** they **already used** in **video cassettes**

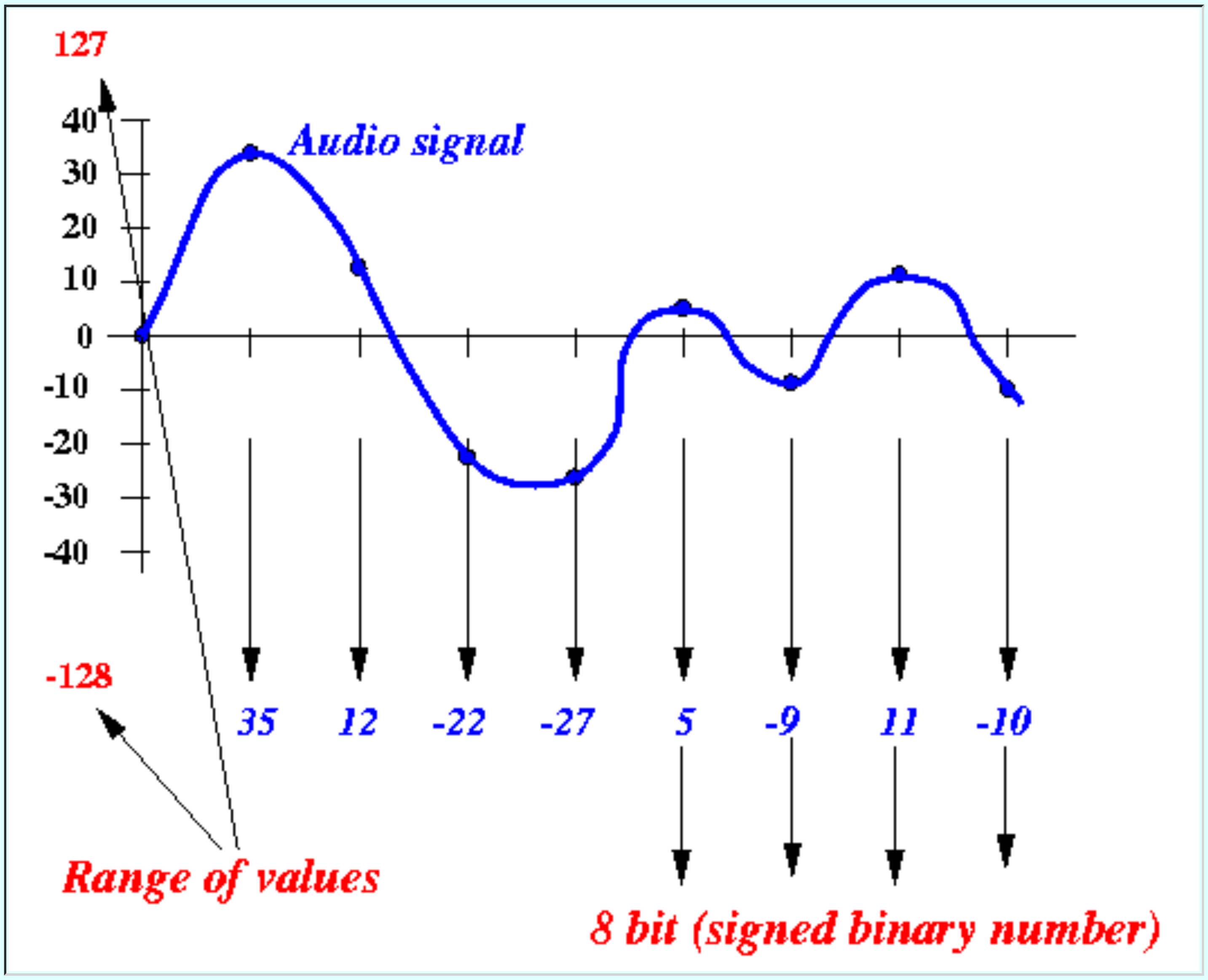
    Reference: click here

- **Encoding CD quality audio**

  - Encoding **standard quality** audio:

    - ***Normally***, an **audio sample** (= the **amplitude *value*** of the **audio signal**) is encoded using **8 bits** (as a ***signed binary number***)

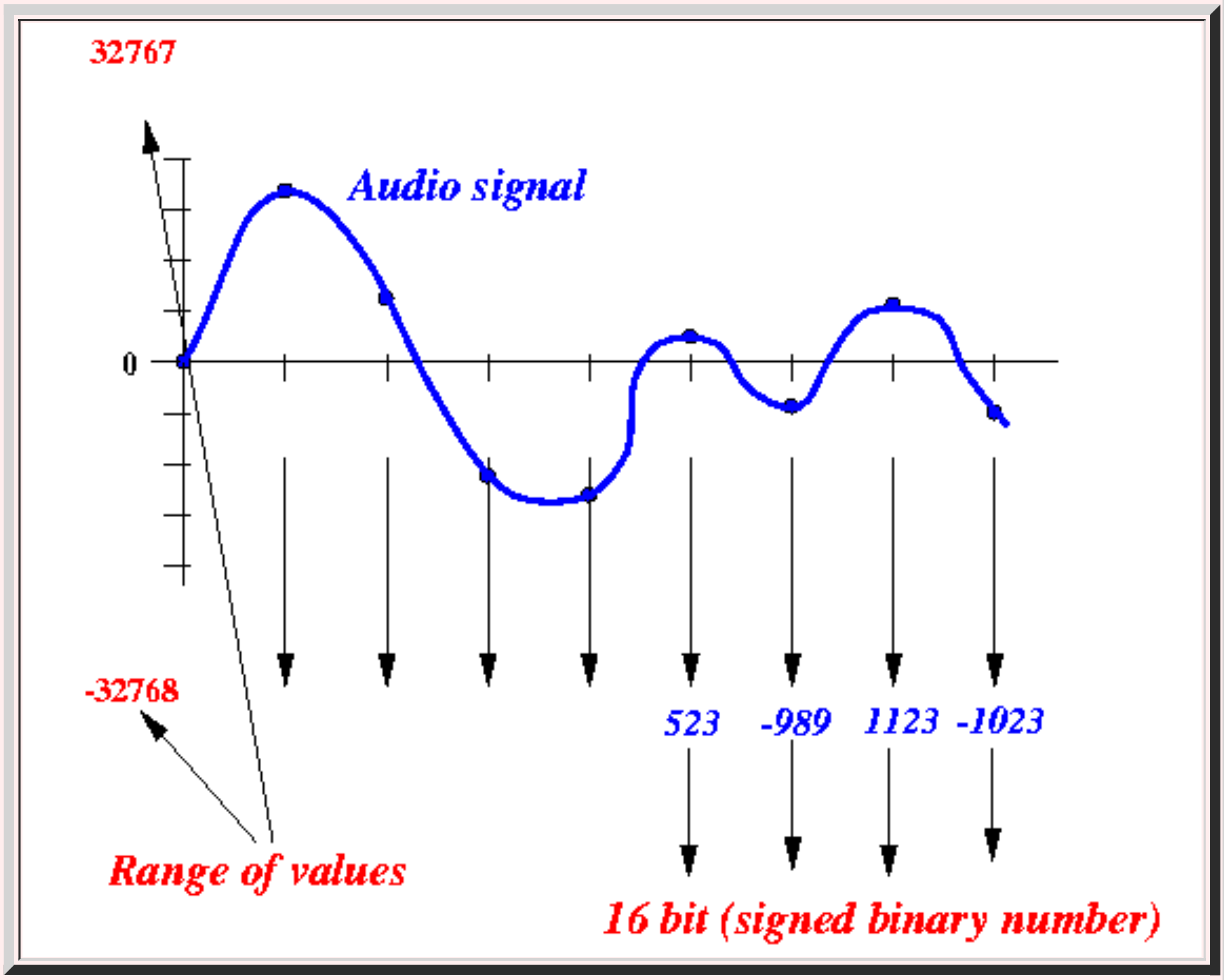
      Example:

  - Encoding **CD quality** audio:

    - In **CD quality audio**, an **audio sample** is encoded using **16 bits** (***signed binary number***)

Example:



(**Result**: ***highly* accurate** representation of the **original audio signal**, but will require ***more* storage**)

- **Data rate of CD-quality audio**
  - **Data *rate* of *mono* CD quality audio**:

```
Sample Rate = 44,100 Hz   (i.e., we sample 44,100 times in 1 sec)

    ====> 44,100 samples in 1 sec

1 sample is encoded using 16 bits

    ====> 44,100 × 16 = 705,600 bits in 1 sec

---------------------------------------------------------------

Data rate of (mono) CD quality audio = 705,600 bits/sec
```

  - **Data *rate* of *mono* CD quality audio**:

```
2 channels used for stereo
```

====> 2 × 705,600 bits = 1,411,200 bits/sec

Application of the Nyquist Rate: Digital Telephone

- **Digital telephone**
  - ***Telephone quality* audio:**

    - **Human ear** can hear frequencies: **20 - 20,000 Hz**....

      But:

      - You do **not** need the ***full* range** to **understand** what someone is saying...

    - **Understandble conversions** can be **carried** using the following **range of frequencies**:

      - **300Hz - 3400 Hz**

- **Nyquist rate for "telephone" quality audio**
  - ***Maximum* frequency** of **"telephone" quality** audio:

    - **3400 Hz** !!!

  - **Nyquist sampling rate** for **"telephone" quality** audio:

```
Nyquist Sampling Rate (telephone quality) = 2 × MaxFrequency
                                          = 2 × 3400 Hz
                                          = 6800 Hz

  i.e.: Minimum sampling rate = 6800 times per second.
```

- **Sampling rate used by telephone companies**
  - **Sampling rate** of **(digital) telephone**:

- Today's (digital) telephone **samples** the voice at:

  - **8000 Hz**

    I.e.: 8000 samples in 1 sec or **1 sample in 1/8000 sec**

- **Encoding "telephone" quality** audio:

  - **Each sample** is **encoded** using **signed binary value** of **8 bits**

- **Data rate of "telephone" quality audio**

  - **Data rate** of ***telephone quality*** **audio:**

```
Sampling Rate = 8000 Hz

     ====> 8000 samples in 1 sec

1 sample outputs 8 bits

     ====> 8000 × 8 bits = 64000 bits in 1 sec

-----------------------------------------------------

Data rate of telephone quality audio = 64,000 bps
```

- **Digital telephone**

  - **Fact:**

    - The **telephone company** reserves a **64,000 bps communication channel** for each **phone conversation**

    However:

    - The telephone company **"steals" one bit** from **every byte (8 bits)** for **administration purposes**

      - **E.g.**, send the **caller ID** to the **phone**

- **Effective data rate** of a **telephone channel**:

  - **effectively**, a **telephone channel** transmits:

    - **7/8** × **64,000 bps** = **56,000 bps**

    of ***real* audio data**

- **A *gadget* from the past**

  - The **equipment** that allows you to **transmit data** over a **telephone connection** is:

    - a **modem** (**modulator/de-modulator**)

  - Due to the **channel capacity** of a **telephone connection** (see **above** !!!), the **modem speed** is **limitted** to **56 kbps** !!!

Modem is a thing of the **past** because **telecom companies** now sells ***Data* Services** to their customers.

Encoding/decoding *digital* data using *digital* signals

- **Digital data**
  - **Digital data:**
    - **Digital data** = ***discrete* values**

      Example:

      - **12**, **45**, **-67**
      - **3.14**, **4.12**, ...

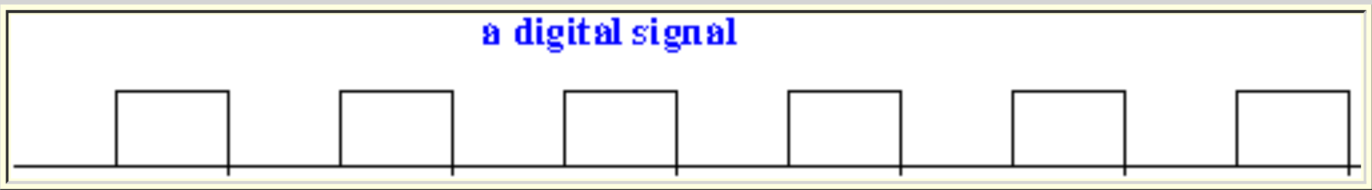
  - **Fact:**
    - **Digital values** can be **converted** into a ***binary* representation**

      Example:

      - **12** **<->** **00001100**

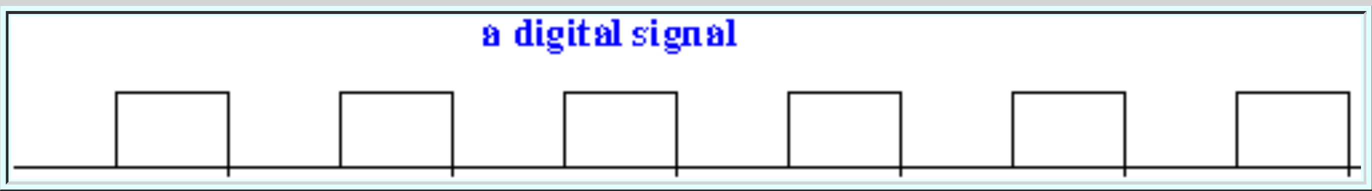
- **Digital signal**
  - **Digital signal:**
    - **Digital signal** = a **signal** with **finite number** of ***discrete* levels**

      Example: a ***two* level digital signal**

- **Digital data transmission with digital signals**
  - **Transmitting *digital* data using *digital* signal:**
    - **Digital data: 0101010101**

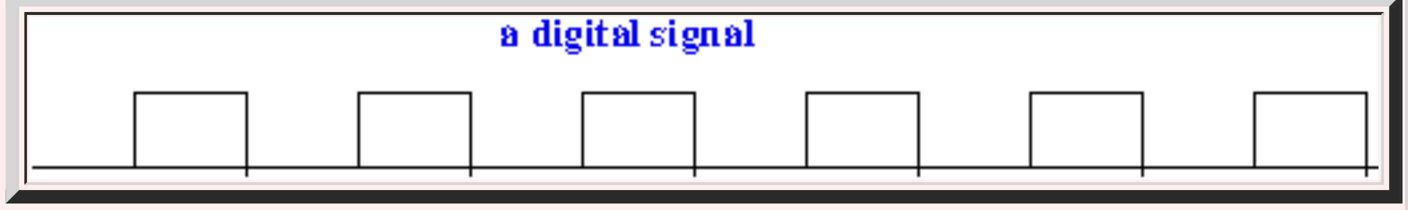
    - **Digital signal** to relay this **digital data**:

- **The missing ingredient in digital transmission**

  $64,000 questions:

- **Suppose** you **receive** this **digital signal**:



Question:

- **What** was the **data transmitted** ???

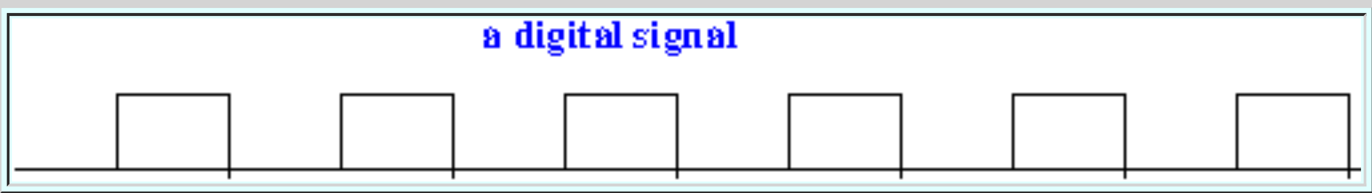
Obvious answer:

```
010101010101
```

Is this the ***only* answer** ?????

- **Decoding a digital signal**
  - ***Illustrative* example:**

- **Suppose** you **received** the following ***digital* signal:**

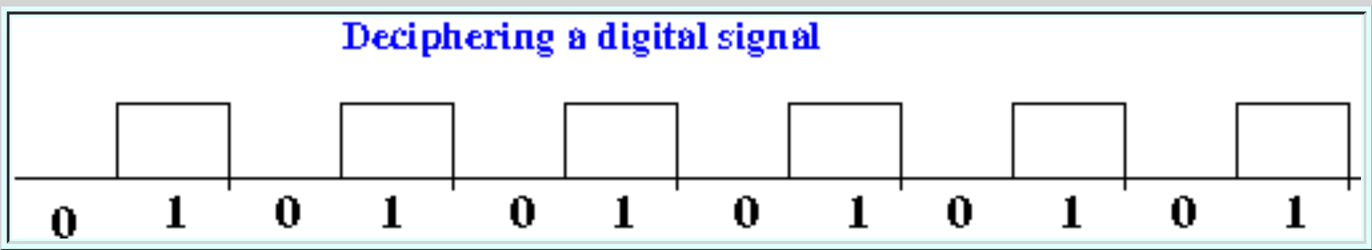


Question:

- What was the **digital data** that is **encoded** by this **signal** ???

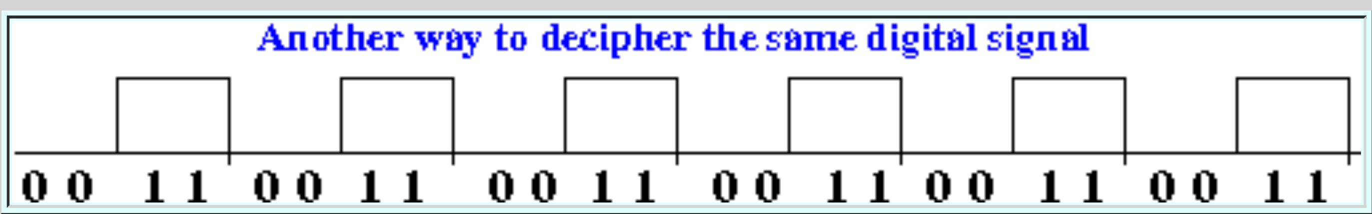
- An ***obvious* answer** is **010101010101**

because:



- An ***not so* "obvious"** answer is **001100110011001100110011**

because:

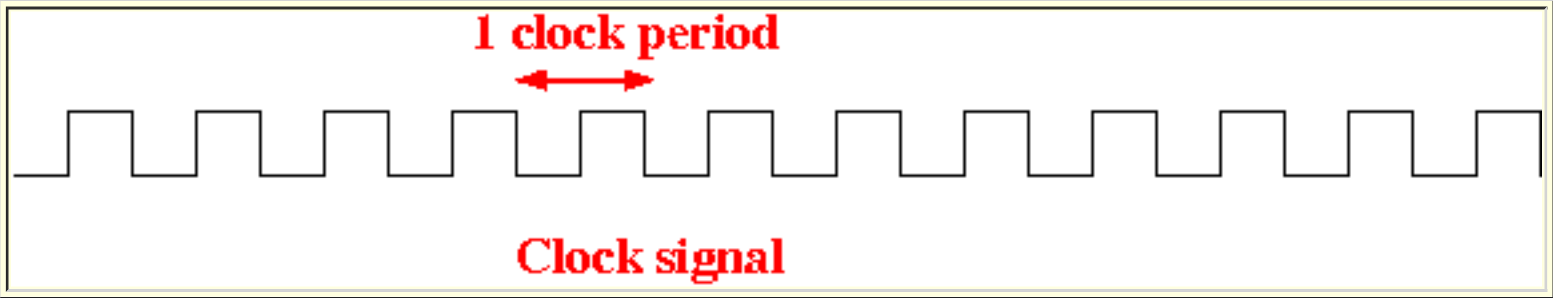


- **Answer** to the **$64,000 question:**

- You **need** to **know** the **transmission rate** (of the **digital signal**) in order to **decode (= determine the result)** a **digital signal** !!!!!

- **Clock signal**
    - **Fact:**
        - **Transmission rate (= speed)** in **electronic devices** are **controlled** by a ***clock* signal**
    - **Clock signal:**
        - **Clock signal** = a **periodic wave** of **0,1** transitions at a **fixed *frequency***

          Example:

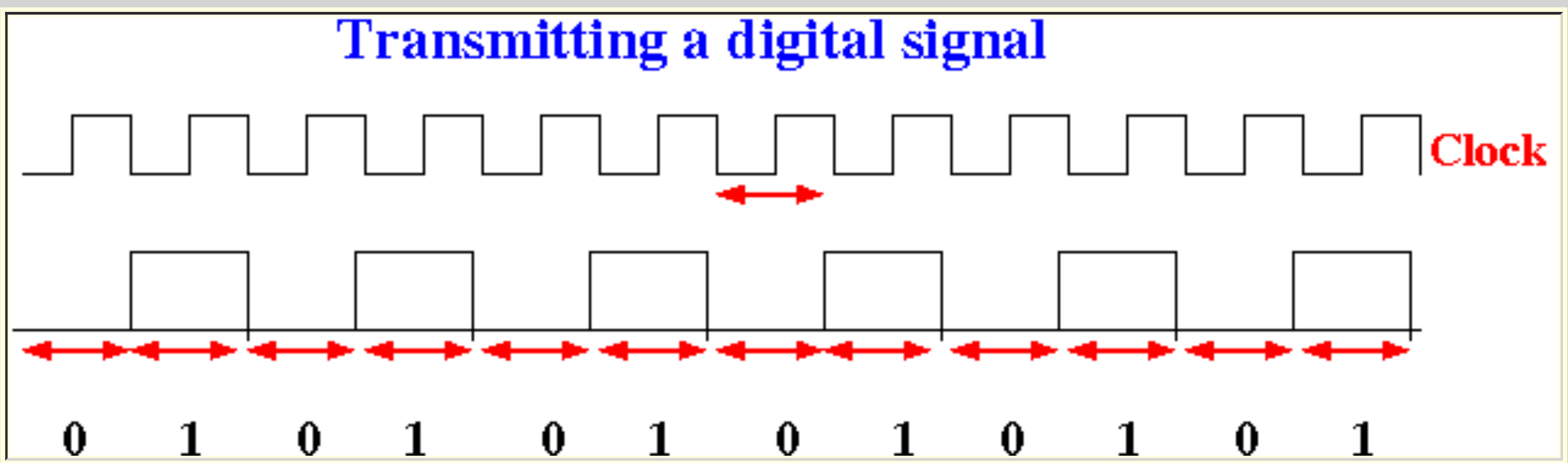
    - **Clock signal generation circuits:**
        - **Clock generator** = a **circuit** that produces a **timing signal** for use in **driving (synchronizing)** a **circuit's operation**.

          See: [Wikiopedia]()
- **Transmitting and decoding digital signals**
    - **How to** use a **clock** in **digital communication**:
        - **Requirement**:
            - **Both** the ***sender*** and the ***receiver*** **must** use a **clock** with the ***same* frequency**

          I.e.:

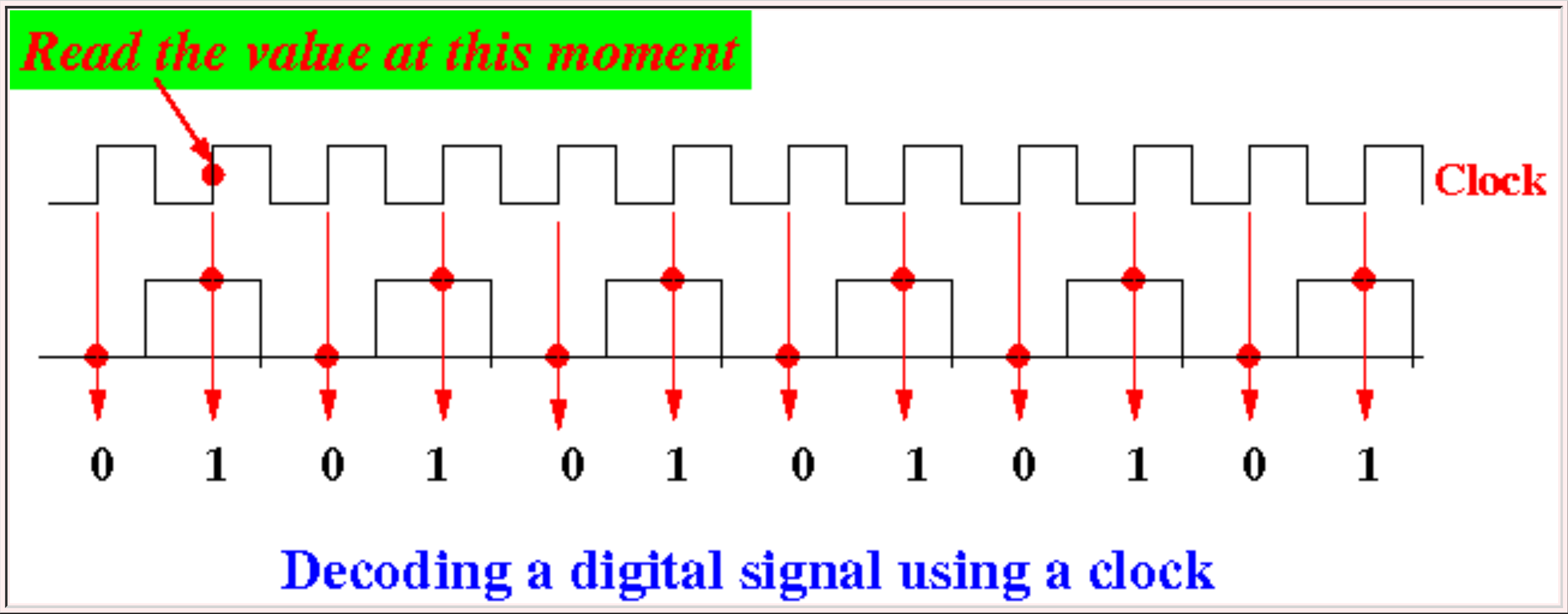
            - **Sender** and **receiver** will **operate** at the ***same* speed** !!!
        - **Transmission method** used by the **sender:**
            - **Transmit** the **signal (= 0 or 1)** for **1 clock period**

              Example:

- **Decoding (= reception) method** used by the **receiver:**
  - **Read** the **received signal** at the ***middle*** **point** of a **clock** ***period***

    Example:



- **Very important:**
  - **Sender** and **receiver** that uses **digital signaling** to **communicate** with each other ***must*** use the ***same*** **clock frequency** !!!

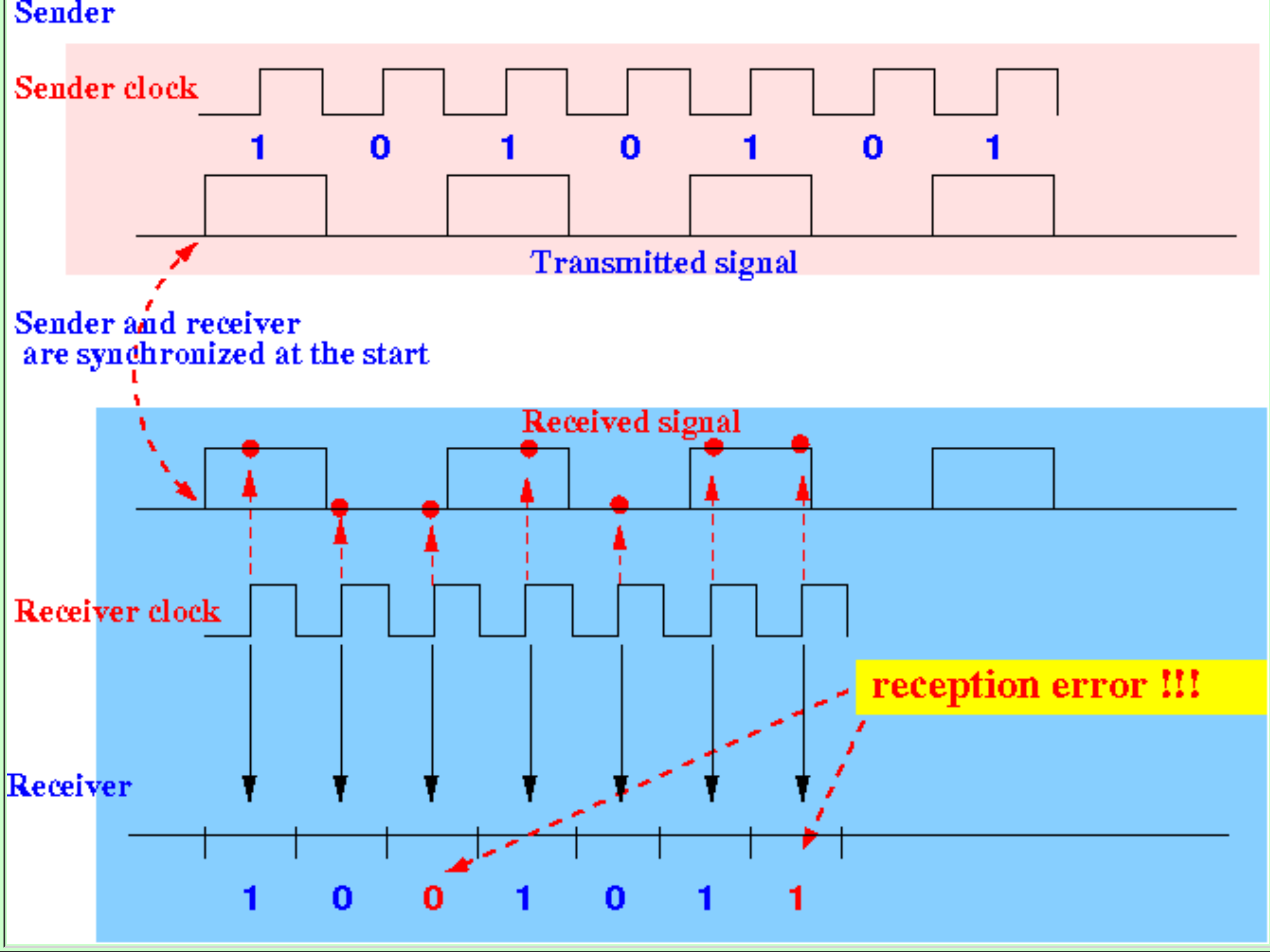
    (Otherwise, you will have ***reception*** **errors** !)

The problem of *clock drift*

- **Click drift**
  - **Fact:**
    - ***No* 2 digital clocks** will ever run on the ***same* frequency**
      - E.g., **temperature** can **affect** the **frequency (= speed)** of **digital clocks** !!!
  - **Fact of (normal) life:**
    - **Two "identical" clock signals** (used in **computer communication**) will run at ***slightly* different speeds**
  - The **difference** in **clock speed** is called:
    - **"drift"**.

- **Consequence of clock drift**
  - **Consequence of clock drift:**
    - **Clock drift** can **cause *reception* (decoding) errors *if***:
      - A ***large* number** of **bits** is being **decoded**

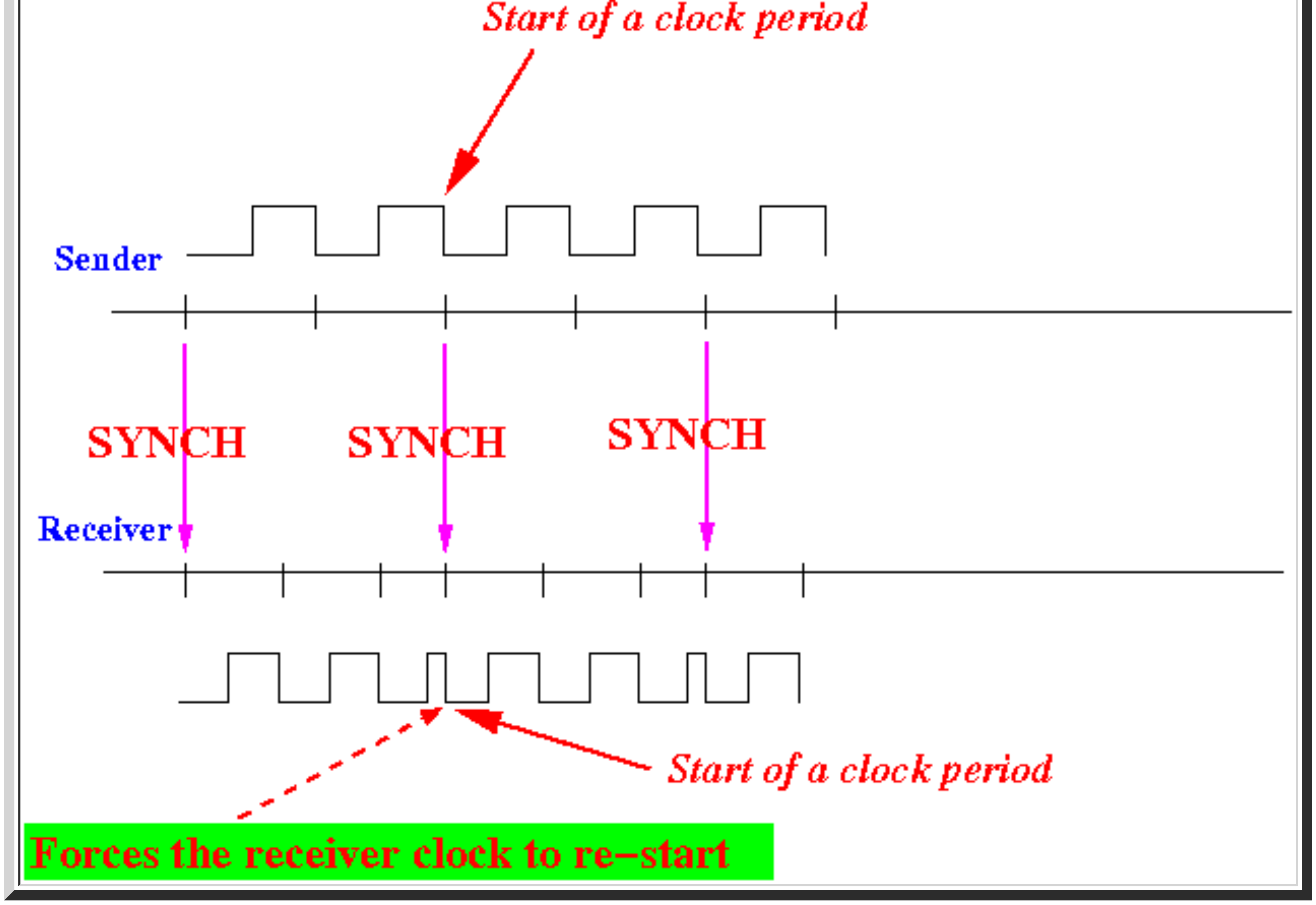
      Example:



- **Receiver clock *(re-)synchronization***
  - **Important conclussion:**

    - The **receiver** must (from time to time) ***re-synchronize* it clock** with the **sender's clock**

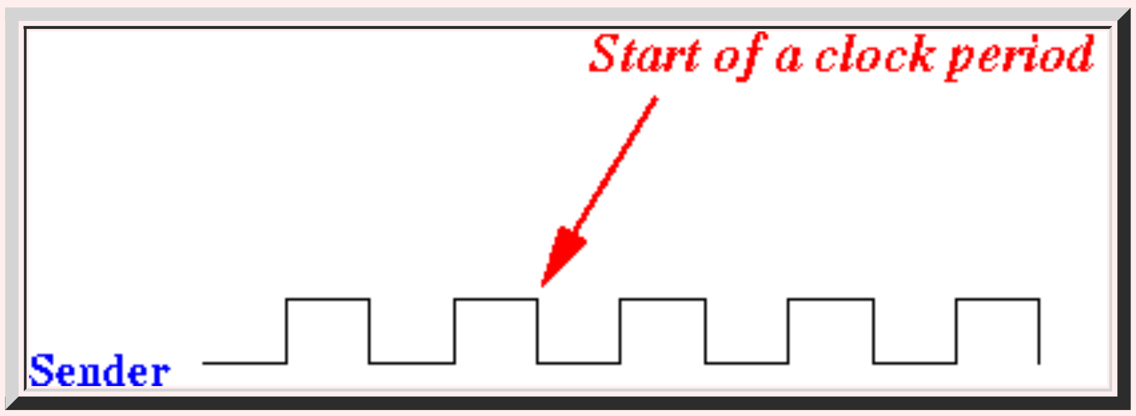
  Example:



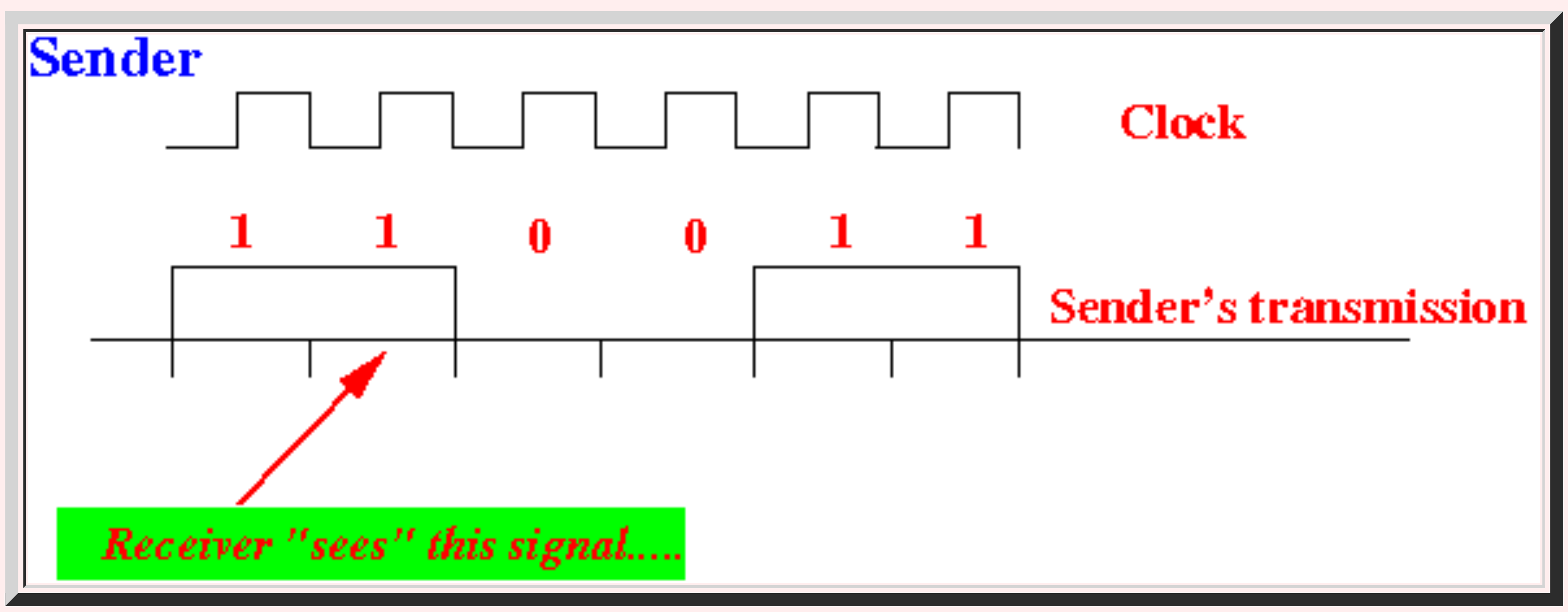
- ***When*** **can you re-synchronize the receiver's clock ???**

  Problem:

  - The **receiver** can **not** **"see"** the **sender's clock**:

  - The **receiver** can ***only*** **"see"** the **transmission** from the **sender**:

  - **This** is what the ***receiver*** will **"see"**:

- ***When*** **can** the **receiver** **perform** a **clock (re-)synchronization** ???

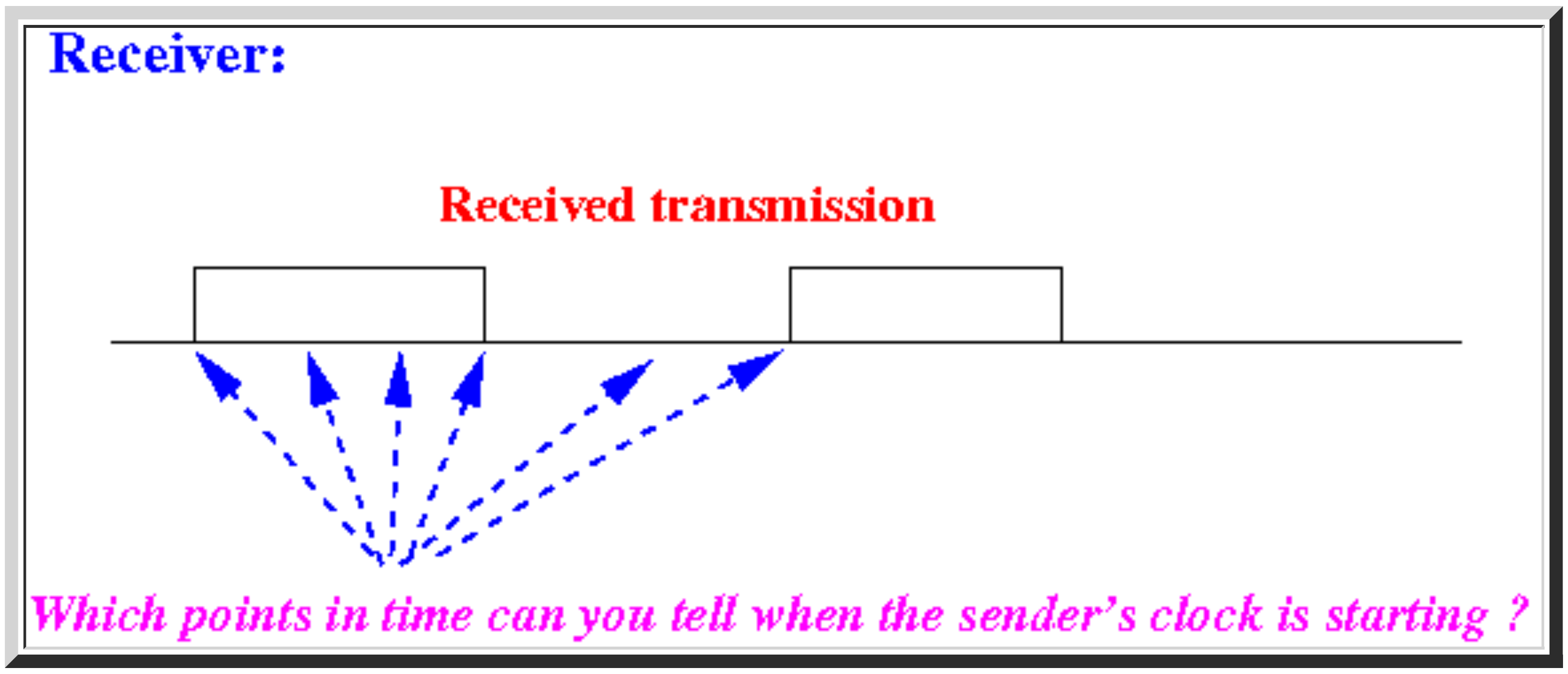
  I.e.:

  - Which **moments** can the **receiver** **know *for sure* what** the **state** of the **sender's clock** is ???

- **$64,000 question:**

  - ***Which*** moment(s) in **time** can you know **for sure** that the **sender's clock** is at the ***beginning*** of a **clock period** ???

  Pictorially:

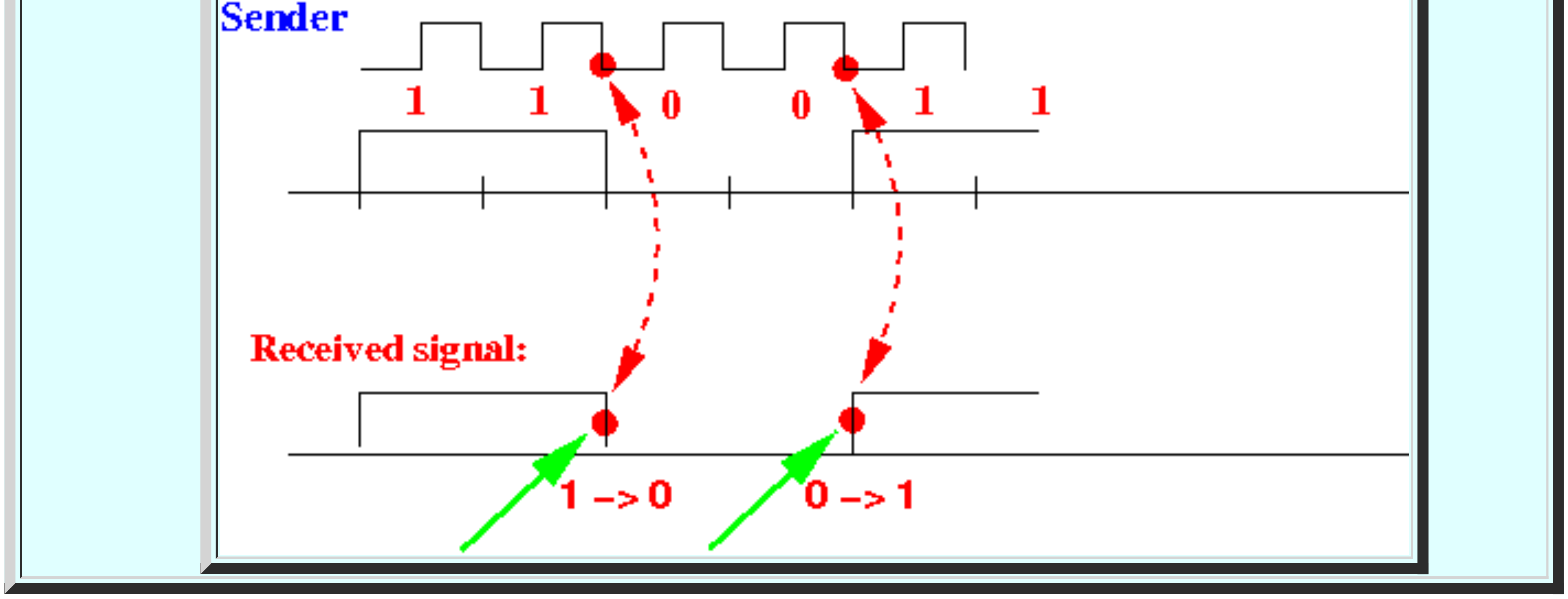
  

  Answer:

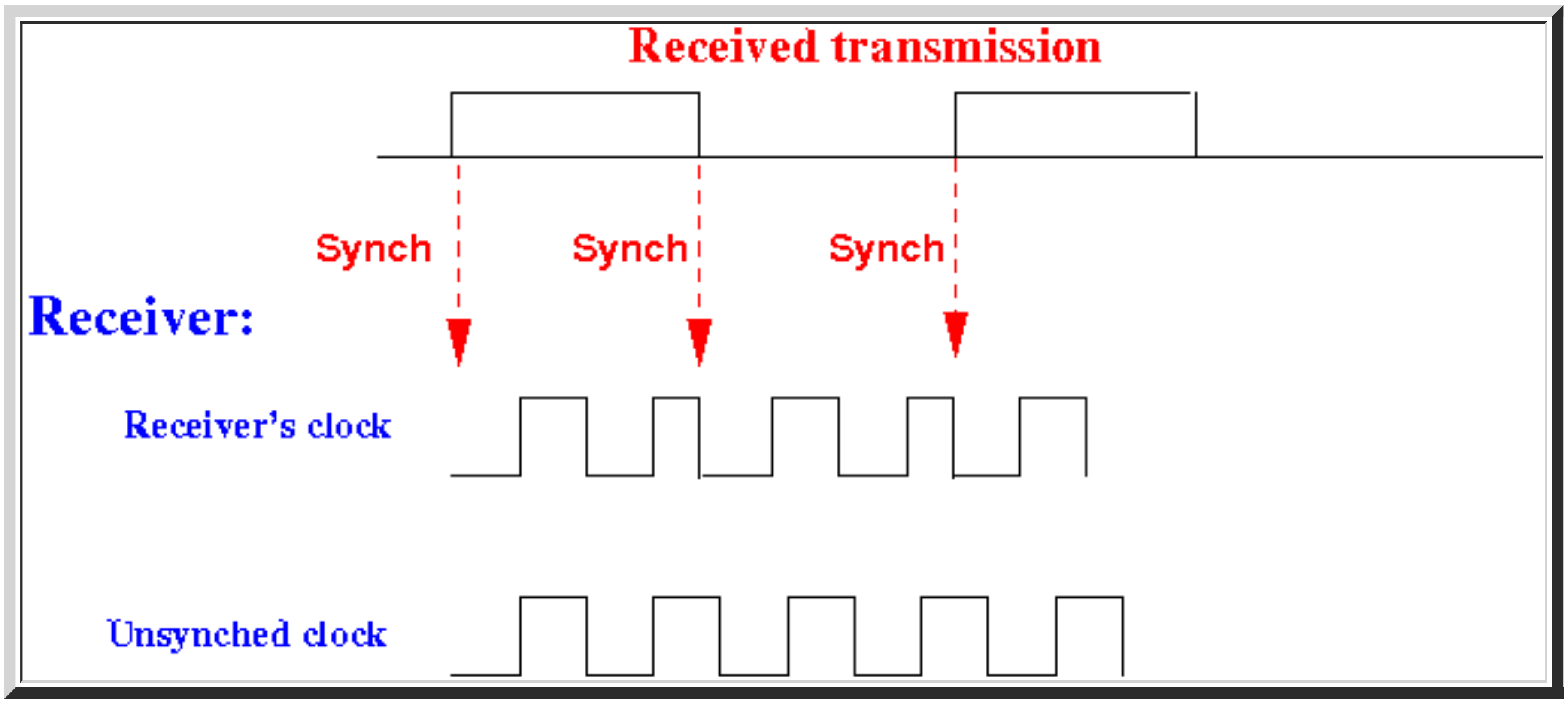
  - **Receiver** can ***re-synchronize*** its **clock** when the ***received*** **signal** makes the following ***transition***:

    - from **0 → 1** **or**
    - from **1 → 0**:

    Graphically:



Example:



- **General practice:**

  - ***Each time*** when the ***received*** **signal** makes a ***transmission***:

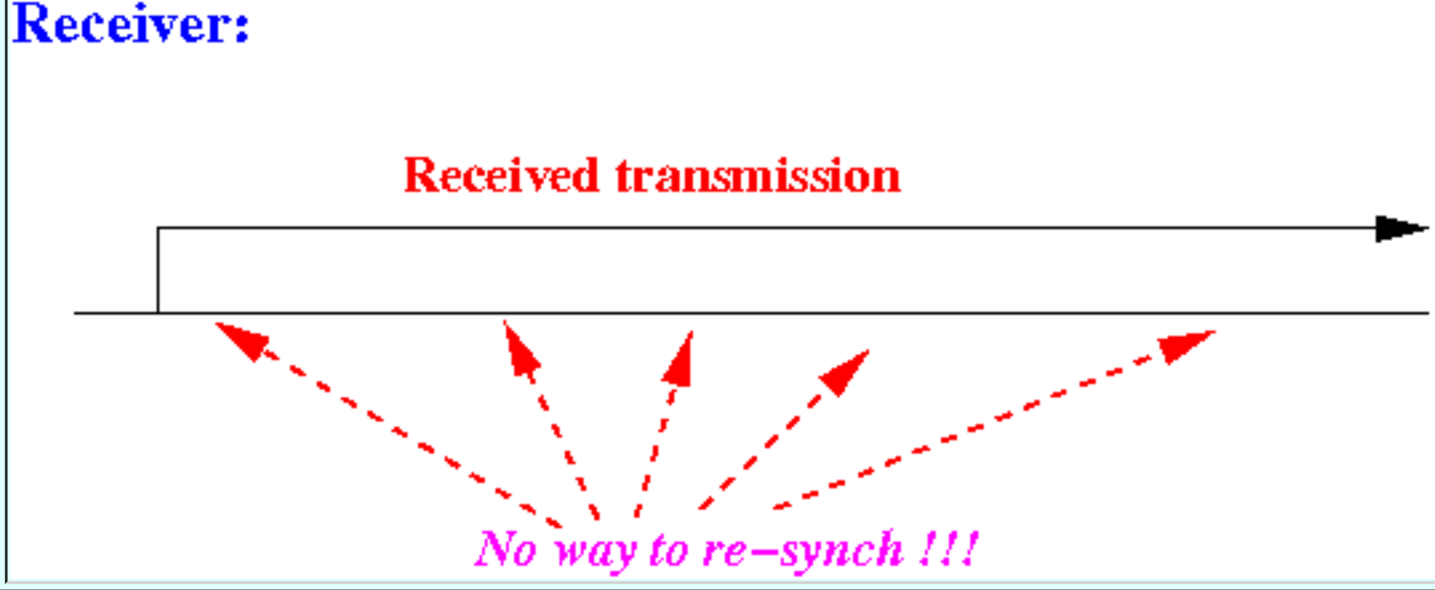
    - The ***receiver*** **clock** is ***re-synchronized*** *!!!*

  (**Re-synchronize** means: **force** the **clock signal** to **restart** at the ***start*** **of a period**)

- ***Wait*** **!!! There is still a problem...**

  - There is ***still*** a **problem**:

    - If the **sender** transmits a ***long*** serie of **0's or 1's**:



the **receiver** can **not** perform **re-synchronize**

- **Solution ???**

    - **More *complicated* transmission encoding** !!!

The NRZ (Non-Return to Zero) transmission code

- **Digital transmission *codes***
  - There are **2 signaling levels** in ***binary* digital transmission**:
    - ***low* (= 0)**
    - ***high* (= 1)**

  - ***However*:**
    - There are ***different* ways** to ***signal* data** using these **2 signal levels** !!

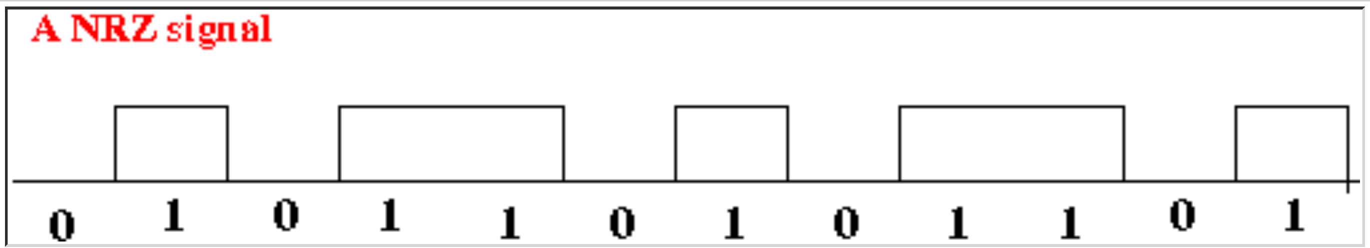
  - We will look at some **commonly used** schemes
    - **Each signaling scheme** has its **strengths** and ***also***, it's **weakness(es)**

      (The **law of "conservation of misery"** --- Herman Bavinck)

- **The Non-return-to-zero (NRZ) code**
  - The **Non-return-to-zero (NRZ) encoding scheme**:

    (It's the ***most* intuitive** code :))

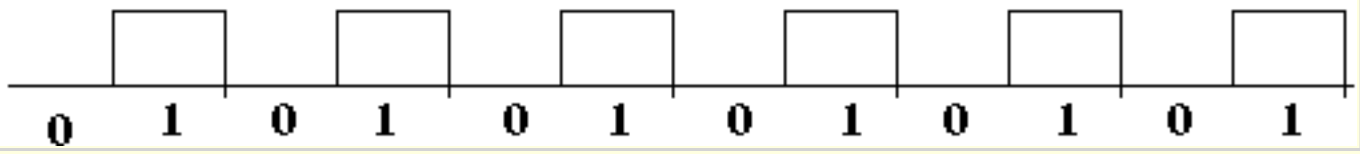
  - **Example:**



  - **Advantage:**
    - The ***highest* frequency** of the **transmitted signal** is **relatively *low***

      Explanation:
      - The **signal** will **vary** at its ***fastest* rate** when you transmit as string of **0101010101....**:

Fastest varying NRZ signal:

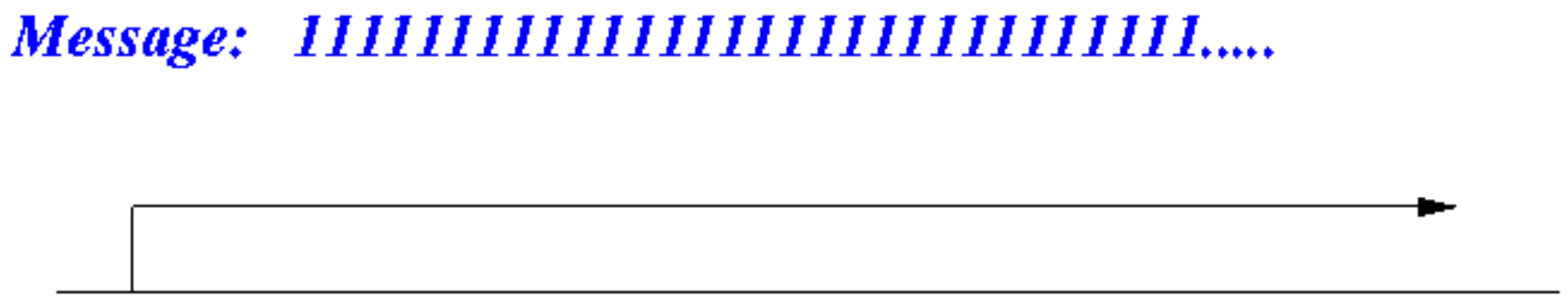


- Compare to ***other* signalling codes** (that you will learn ***later***), this **signal form** varies at the ***lowest* rate possible** !

- **Disadvantage:**

  - There are **situations** where the **receiver *cannot* synchronize** its clock with the **sender's clock**:

    Example:

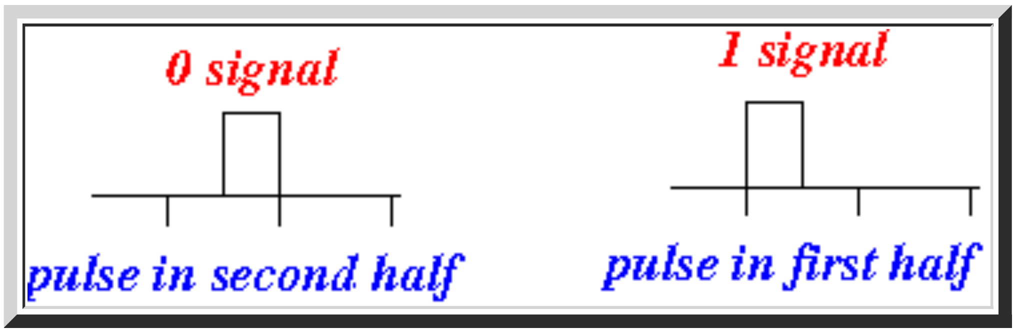
    (There is ***no* transmission** in the signal so the **receiver** can **not perform re-synchronization** !)

- **Common Practice:**

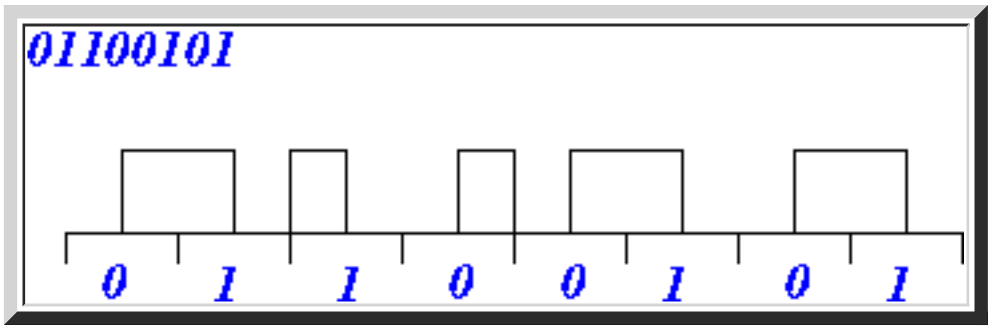
  - Because of its ***shortcoming***, the **NRZ** code is ***only* used** for **very *short* transmissions** (e.g., transmit **1 byte (= 8 bits)**)

The *Manchester* transmission code

- **Manchester encoding: *embedding* the clock in the signal**
  - The **Manchester transmission *code*:**

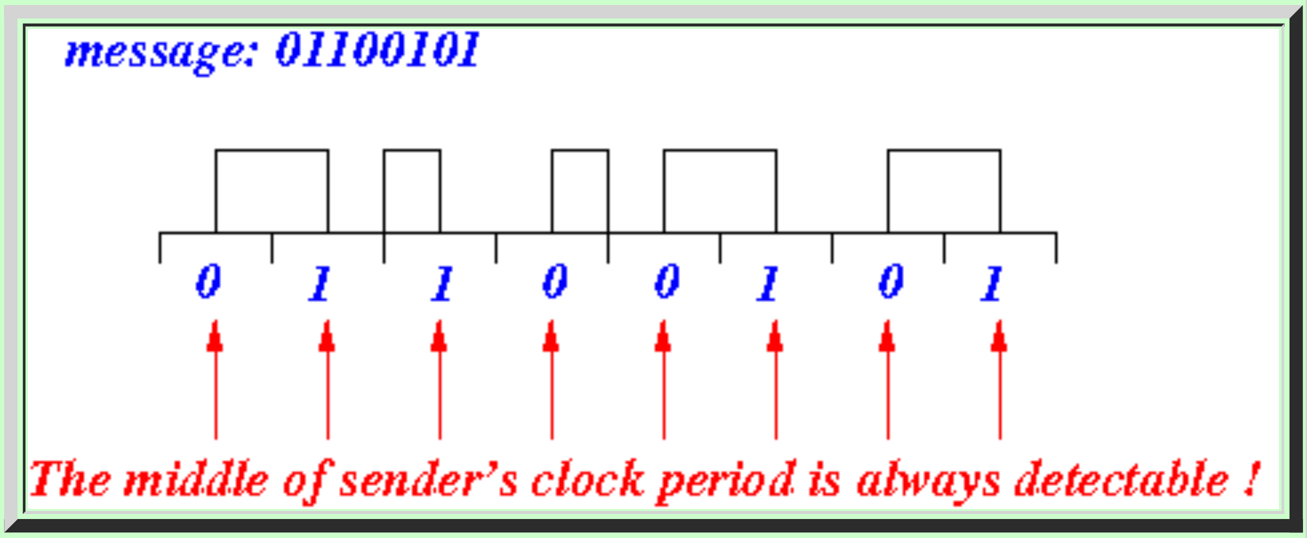


  - **Example:**



  - **Advantage:**
    - The **receiver** can ***always* synchronize** its **clock** at ***every* transmitted bit**:

      Reason: there is a ***transition*** at the **middle** of the **signal** (which is ***detectable***)



    - **Therefore:**
      - The **Manchester code** is ***ideal*** for ***very long* digital**

transmissions (packet length of **several Kbytes**).

- **Practice in real life:**

  - **Ethernet packets** can be as long as **1500 bytes**

    - **Ethernet** uses the **Manchester code** !!!

- **NOTE:**

  - The **synchronization operation** using **Manchester code** will **reset** the ***receiver's clock*** to the ***middle*** of a **clock period**

    (It's easy to do, take **CS355** if you want to learn more about this).

- The **ability** to ***synchronize*** **clocks** does **not** come ***cheaply***....

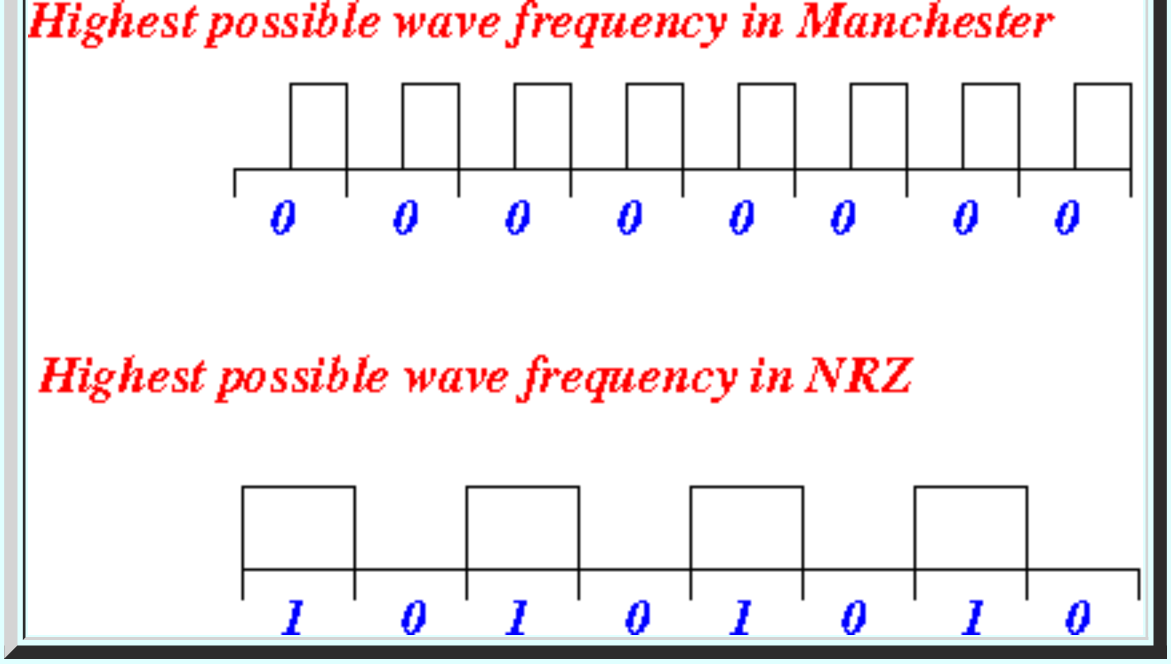
- **Disadvantage:**

  - The ***highest*** **frequency** of the **transmitted signal** is **relatively *high***

    In fact:

    - The **highest frequency** used by the **Manchester scheme** is ***twice*** as large as the highest frequency used in the **NRZ method**

  - **Example:** the **highest frequency** signal using **Manchester code** is achieved when you transmit a string of **00000000...**:



I drew the **highest frequency** of a **signal** using **NRZ** below in the **above figure**.

You can see the **Manchester code** signal has **2 times** the **frequency** as the **NRZ code** signal.

The *4B/5B* trnasmission code

- **4B/5B code: fixing the *synchronization* in NRZ problem cheaply...**
  - The **4B/5B code:**
    - **4B/5B code** = a **very clever way** to provide **synchronization *opportunities*** to the **receiver** by:
      - **Encoding 4 bits** using **5 bits**
      - The **encoding of 5 bits** makes **sure** that there is a **transition** within the **5 bits**
  - **Operation** using **4B/5B code**:
    - The **data** is ***first* transformed** using the **4B/5B encoding scheme**
    - The **encoded result** is **transmitted** using the **NRZ code** as its **basic transmission scheme**

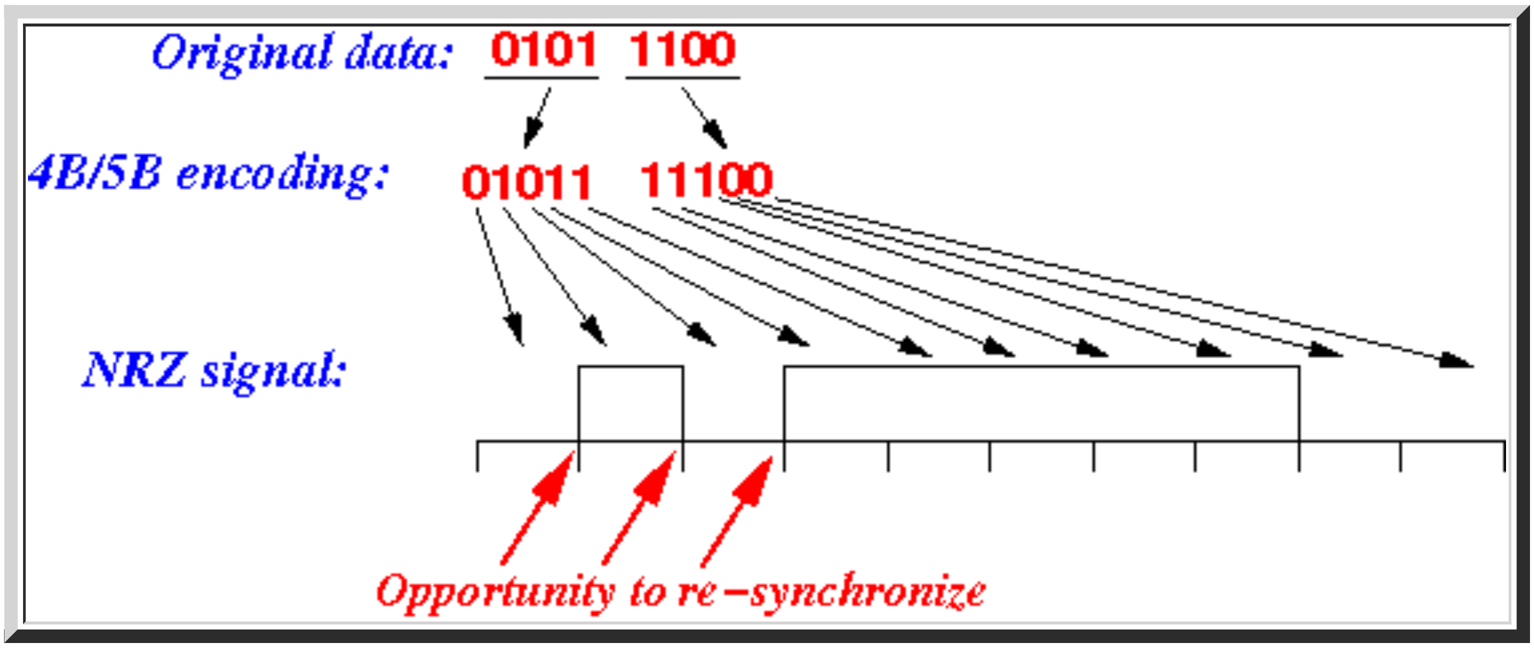
    Note:

    - When the **receiver** received the **transmitted data**, it must use the **reverse mapping** of the **4B/5B encoding scheme** to obtain the **transmitted data**.
  - **4B/5B mapping scheme:**

```
 Original 4 bits          Transformed 4 bits
==================      =======================
    0000                          11110
    0001                          01001
    0010                          10100
    0011                          10101
    0100                          01010
    0101                          01011
    0110                          01110
    0111                          01111
    1000                          10010
    1001                          10011
    1010                          10110
    1011                          10111
    1100                          11010
    1101                          11011
```

1110	11100
1111	11101

- **Example using 4B5B:**



- **Notice that:**

  - After ***transforming*** the original sequence of bits, the **result sequence** will have:

    - ***at most 3*** consecutive **0 bits**

      One way to transmit **3 consecutive 0 bits** is: `0010` (which is transformed to `10100`) followed by `0100` (which is transformed to `01010`)

Original 4 bits	Transformed 4 bits
0000	11110
0001	01001
0010	10100
0011	10101
0100	01010
0101	01011
0110	01110
0111	01111
1000	10010
1001	10011
1010	10110
1011	10111
1100	11010
1101	11011
1110	11100
1111	11101

    - ***at most 8*** consecutive **1 bits**

This will happen when you transmit: 0111 (which is transformed to 01111) followed by 0000 (which is transformed to 11110)

```
 Original 4 bits           Transformed 4 bits
===================     ========================
     0000                        11110
     0001                        01001
     0010                        10100
     0011                        10101
     0100                        01010
     0101                        01011
     0110                        01110
     0111                        01111
     1000                        10010
     1001                        10011
     1010                        10110
     1011                        10111
     1100                        11010
     1101                        11011
     1110                        11100
     1111                        11101
```

Intro: transmitting *digital* data using *analog* signals

- ***Digital* data and *analog* signals**
  - **Digital data:**
    - **Digital data** consists of **discrete values**
      - **Discrete values** can ***always*** be **mapped** onto **integer (whole) numbers**

      Example:

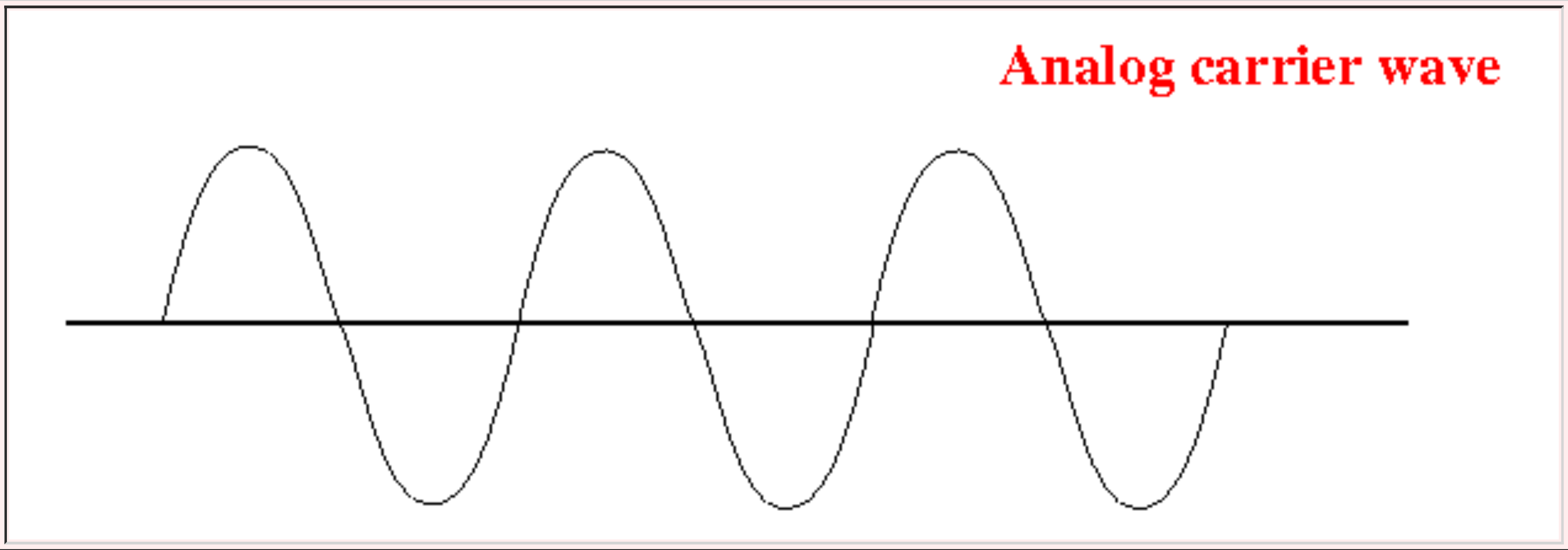
      - **Discrete value set**:
        - {**single**, **married**, **divorced**, **widowed**}
      - **Mapping (representation)** to ***integers***:

```
0   =   single
1   =   married
2   =   divorced
3   =   widowed
```

    - **Integer (whole) numbers** in **computer communication** are **represented** using the **Binary Number System**:

```
0 --> 00
1 --> 01
2 --> 10
3 --> 11
```

  - **Analog signals:**
    - The ***basic* analog signal** is the **sine wave**:



- **Transmitting *digital* data using *analog* signals**
  - **Transmitting *any kind* data** using ***analog* signal** means:
    - **Modulate (change)** the **sine wave** using the ***input* data**
  - **Transmitting *digital* data** using ***analog* signal** means:
    - **Modulate (change)** the **sine wave** to **represent** the values **0 and 1**
  - **Recall**:
    - There are **3 *modulation* methods**:
      - **Amplitute Modulation (AM)**: Change the **amplitude** of the (sine) wave
      - **Frequency Modulation (FM)**: Change the **frequency** of the (sine) wave
      - **Phase Modulation**: Change the **phase** of the (sine) wave

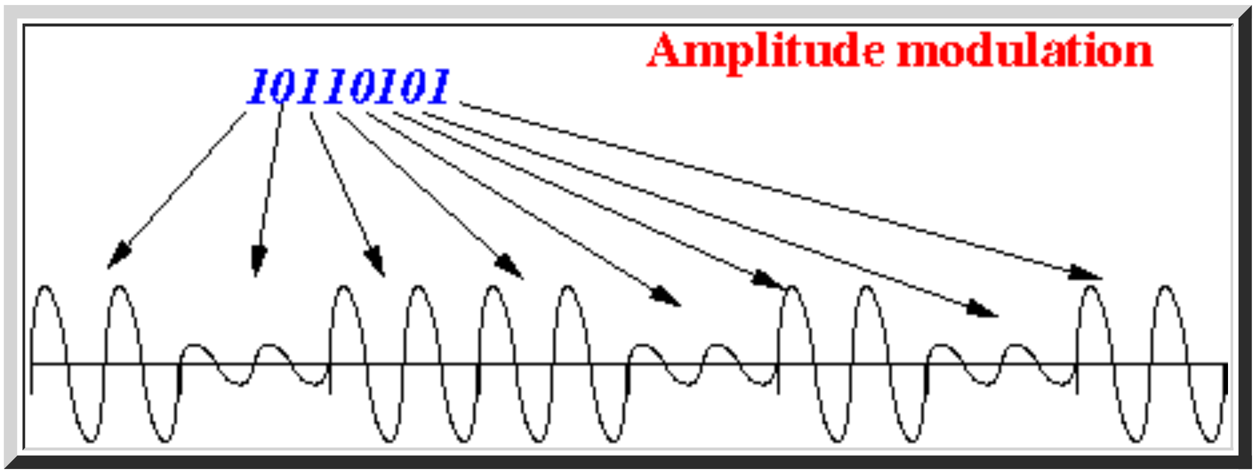
Transmitting *digital* data using *amplitude* modulation

- **Transmitting *digital* data using *Amplitude* Modulation**
  - The **amplitude** of a **wave** determines its **loudness**...

  - **Amplitude modulation:**

    - **0** = transmit a ***softer* signal** for **one time unit** and
    - **1** = a ***loud* signal** for **one time** unit

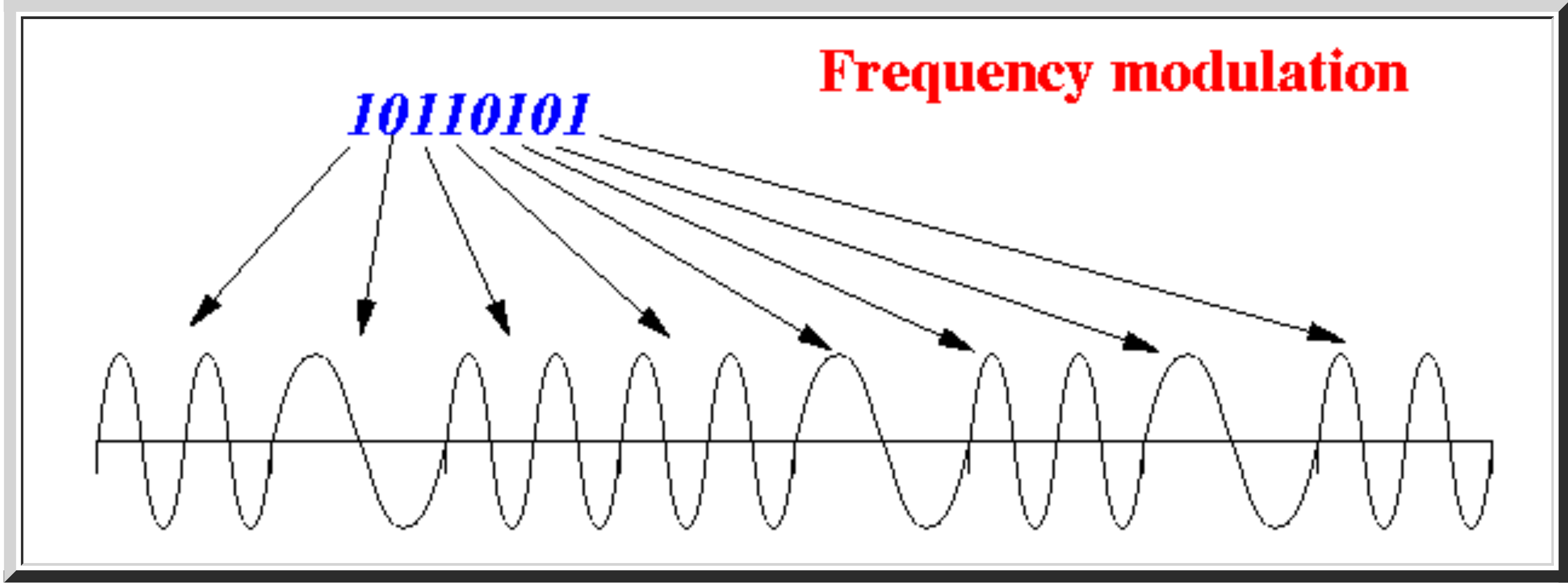
    Example: (in example, **1 time unit** = **2 sine waves**)

Transmitting *digital* data using *frequency* modulation

- **Transmitting *digital* data using *Frequency* Modulation**
  - The **frequency** of a **wave** determines the **pitch** (of the tone)
  - **Frequency modulation:**
    - **0** = a ***lower pitch* tone** for **one time unit** and
    - **1** = a ***high pitch* tone** for **one time unit**

    Example: (in example, **1 time unit** = **2 sine waves**)

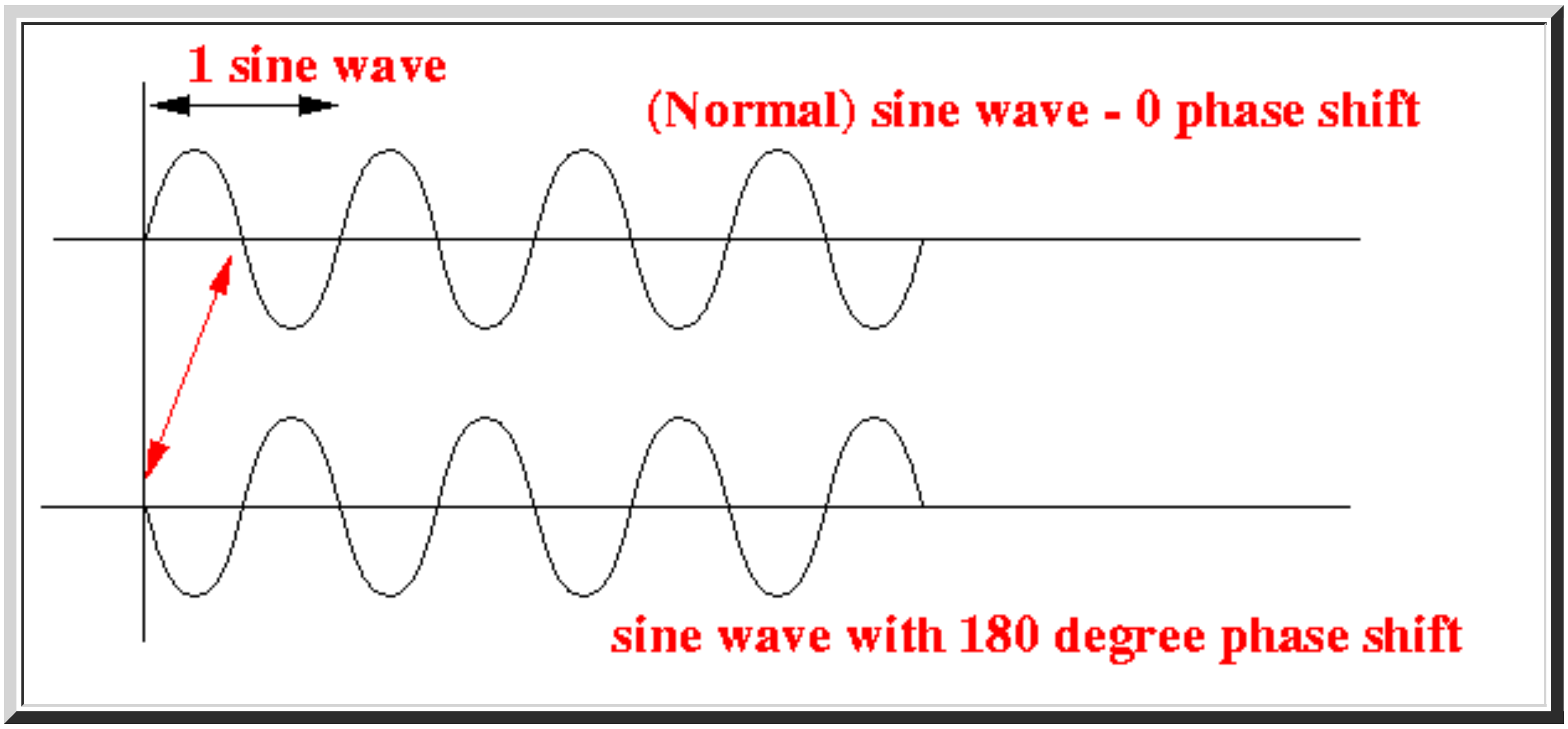
    

  - **Cool demo** on **YouTube** on **Frequency Modulation**: [click here]()

Transmitting *digital* data using *phase* modulation

- **Transmitting *digital* data using *Phase* Modulation**
  - The **phase** of a **(sine) wave** is the **shift** in the **x-axis direction**

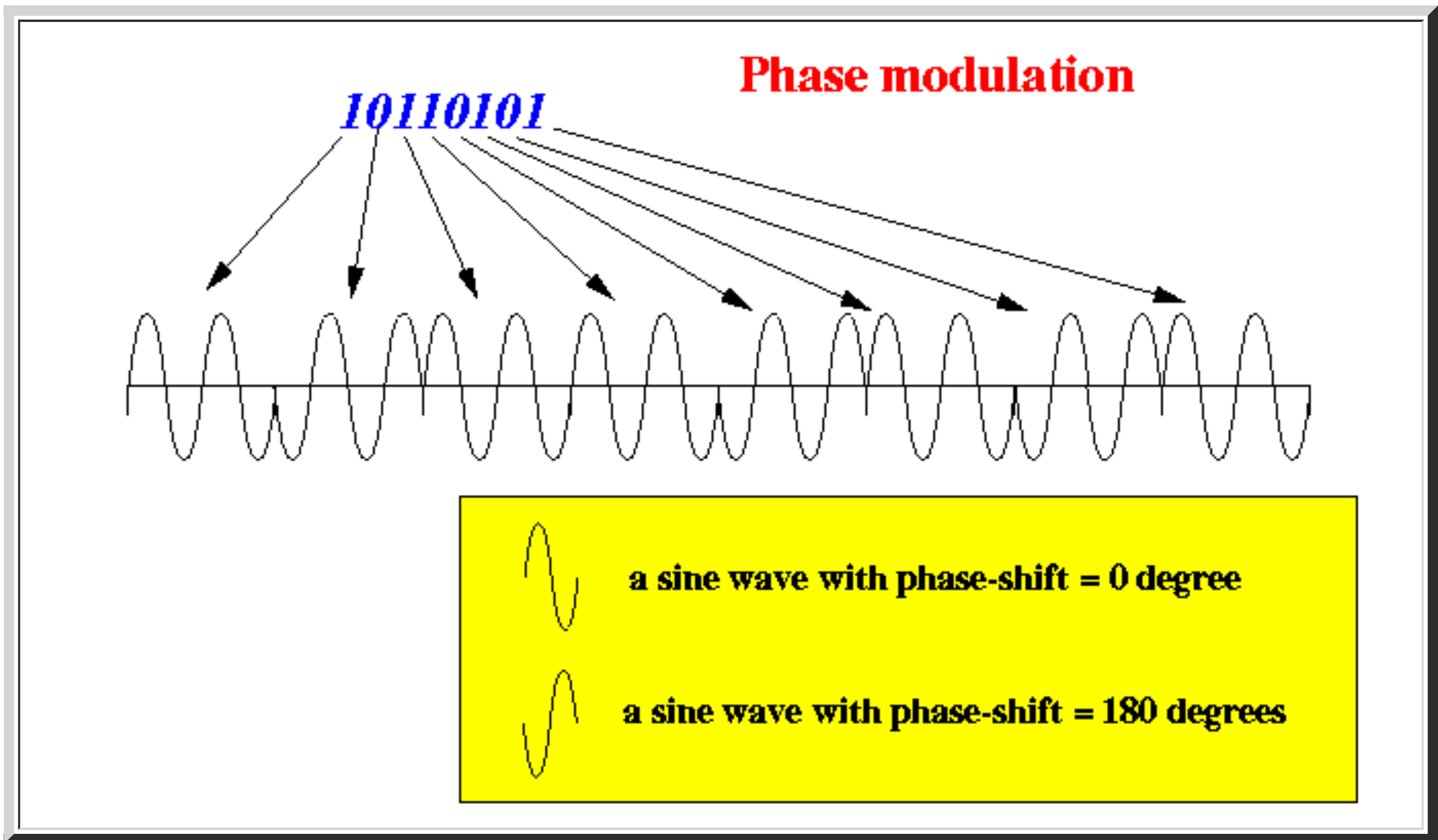
    Example:



  - **Phase modulation:**

    - **0** = a **sine wave** with **0 phase shift** for **one time unit**
    - **1** = a **sine wave** with **180 degree phase shift** for **one time unit**

    Example:



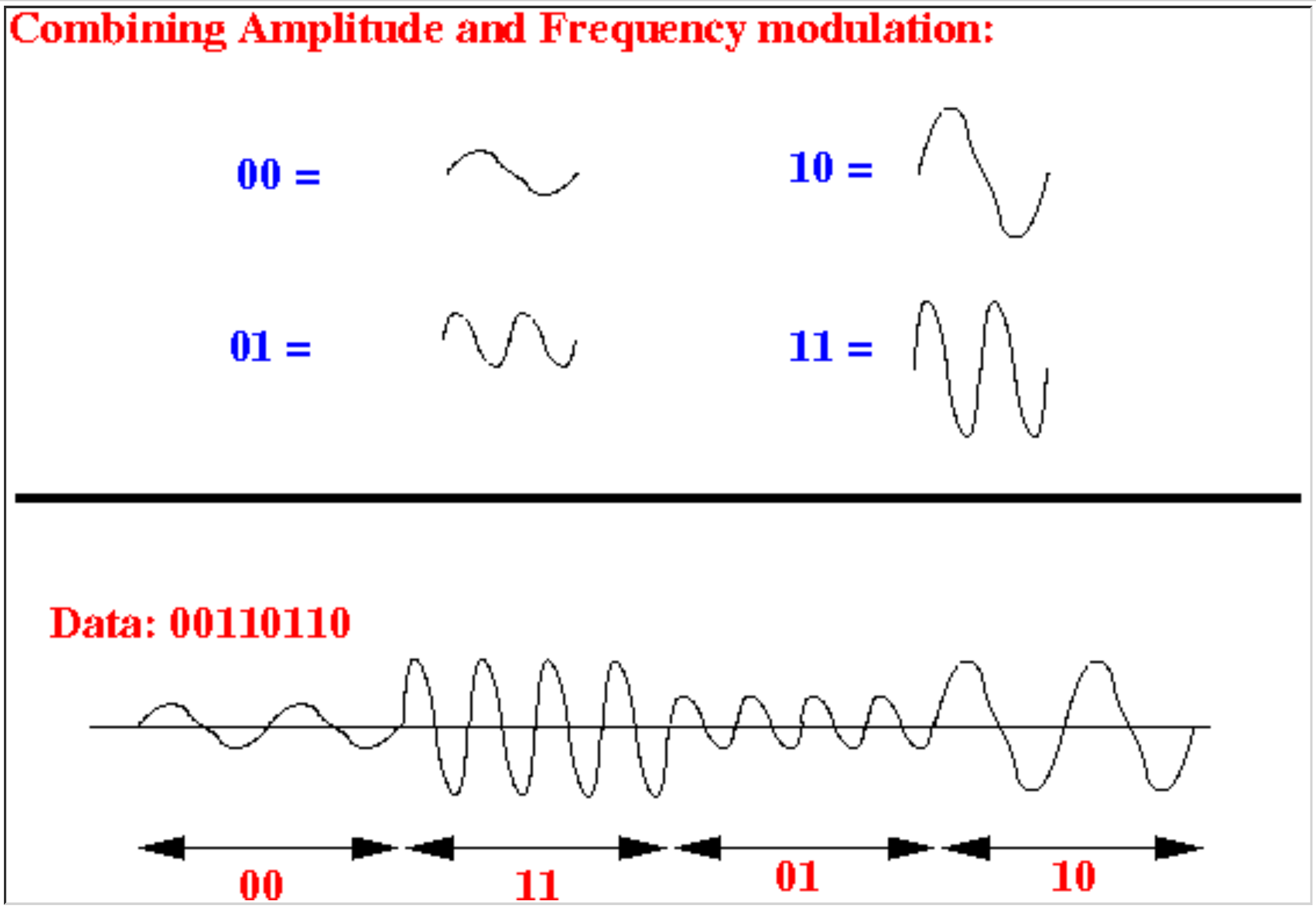
Transmitting *digital* data using a *combination* of modulation techniques

- ***Combining* modulating techniques in *digital* data transmission**
  - **Fact:**
    - It is **common** to **combine** the above **3 modulation methods** to achieve a ***higher* data rate**

Example: combining **amplitude** and **frequency**

```
00          low amplitude     and  low  frequency
01          low amplitude     and  high frequency
10          high amplitude    and  low  frequency
11          high amplitude    and  high frequency
```

Graphical example:



Notice that:

- We can **now** transmit:

- 2 bits using 1 sine wave

- **Historical fact:**

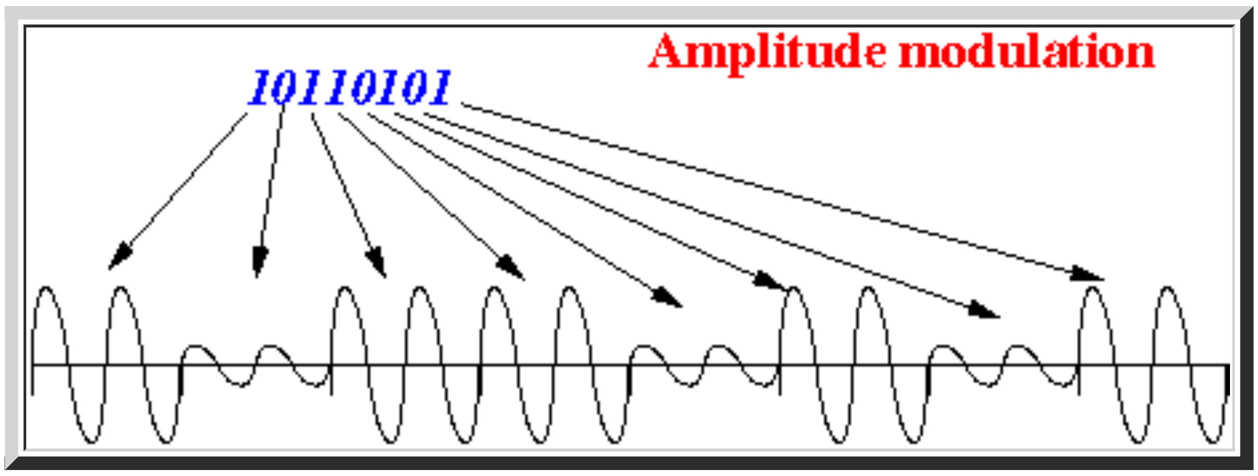
  - ***Modems*** (which was used around **1990**) use a **combination** of all **3 techniques** and can achieve upto **56Kbps**:

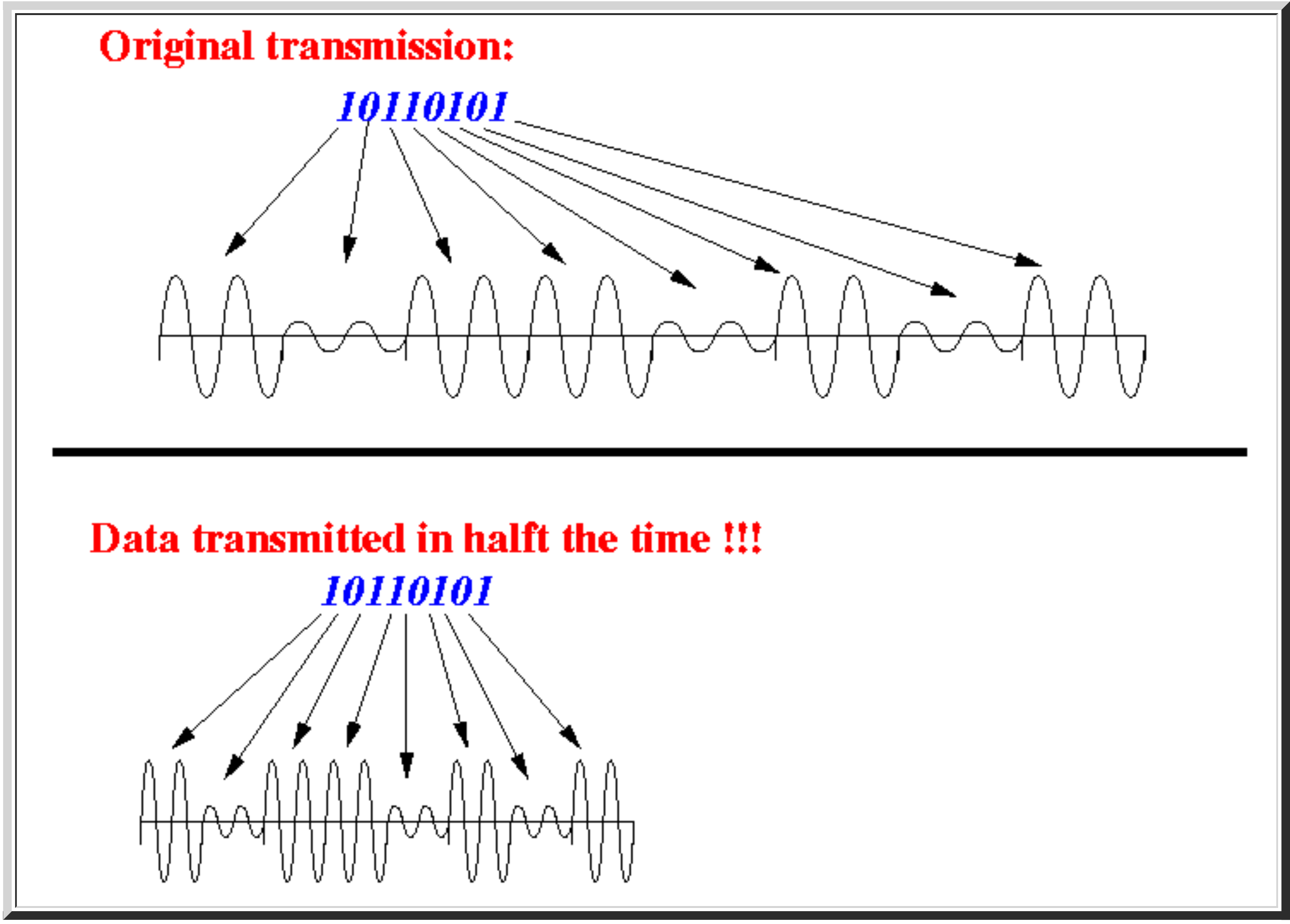
    - which was the ***maximum*** possible **data rate** on a **telephone line**

Bandwidth and *maximum* transmission rate

- **How to transmit data *faster***
  - **Recall**: transmitting **digital data** using ***amplitude* modulation**:



  - **One way** to **transmit** data **faster** is use a ***higher* frequency** signal:



  - **So:**
    - We can **achieve *infinitely* high** data **transmission rates** ???

Obviously* not**... by ***why ???

- **Interesting question....**
  - **Question:**
    - ***Why*** can't we transmit ***tera*** **bytes** of **data** per **second** over - for example - a **copper wire** ???

  - **More *precisely*:**
    - ***What*** **prevents** us from transmitting ***tera*** **bytes** of **data** per **second** over a **copper wire** ???

Speed limitation in *wireless* transmissions

- **Signal used in wireless transmissions**
  - **Wireless** transmission:
    - Always use **electro-magnetic waves**
      - **Note:** ***light*** is ***also*** a form of **electro-magnetic wave** !!!

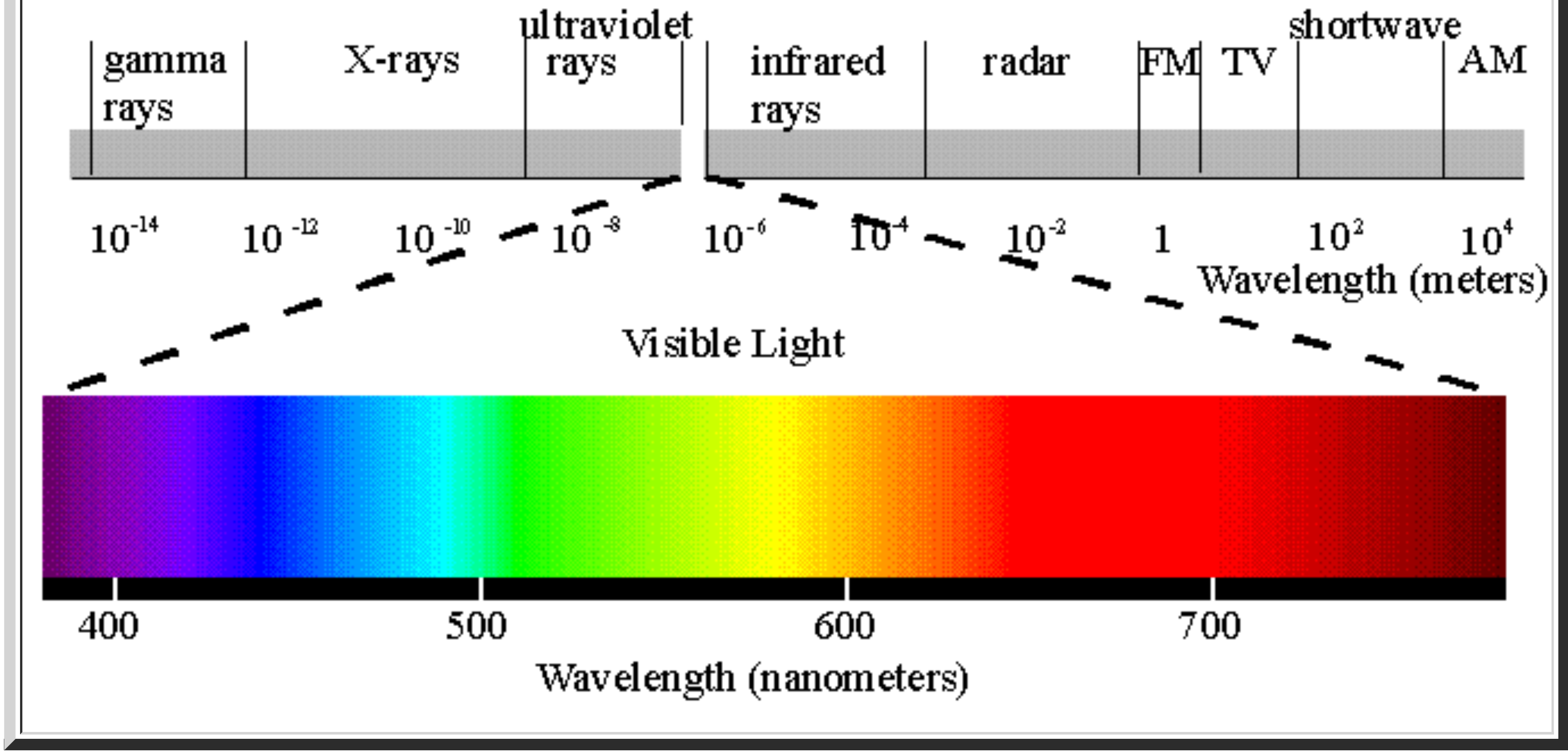
    Example: **radio wave**

- **The *spectrum* of Electro-Magnetic waves**
  - **Electro-magentic waves** goes by **many *different* names**, such as:
    - **Radio wave** (this is what **ordinary people** would think of...)
    - **Light** !!
    - **Micro-wave** (the kind of **electro-magnetic waves** used to **cook**)
    - **X-ray**
    - And so on....

    The ***difference*** between the **electro-magnetic waves** above is:
    - Their ***frequency***....

  - The **frequencies** of **electro-magnetic waves** spans a **very wide range**:



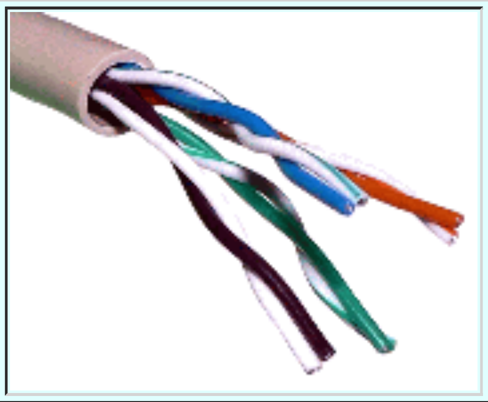
* **Transmission data at *higher* (data) rate**
  * **Fact:**
    * **Higher data rate** transmissions must use a ***high* frequency** **electro-magnetic wave signal**
  * **Consequence:**
    * To achieve ***very high*** **data rate**, we would have to use ***X-ray* or *gamma-ray*** !!!
      * These **rays** are ***harmful*** to **biological entities** !!!!

* **Breaking news....**
  * **Cell phone** and **brain cancer**:
    * Be careful with your **cell phones**:
      * Cellphone radiation can cause cancer: click here

      (**Cellphone** operates at **frequencies** that is ***very* close** to ***micro-wave*** --- the kind of **waves** used to ***cook* food**....

Transmission media

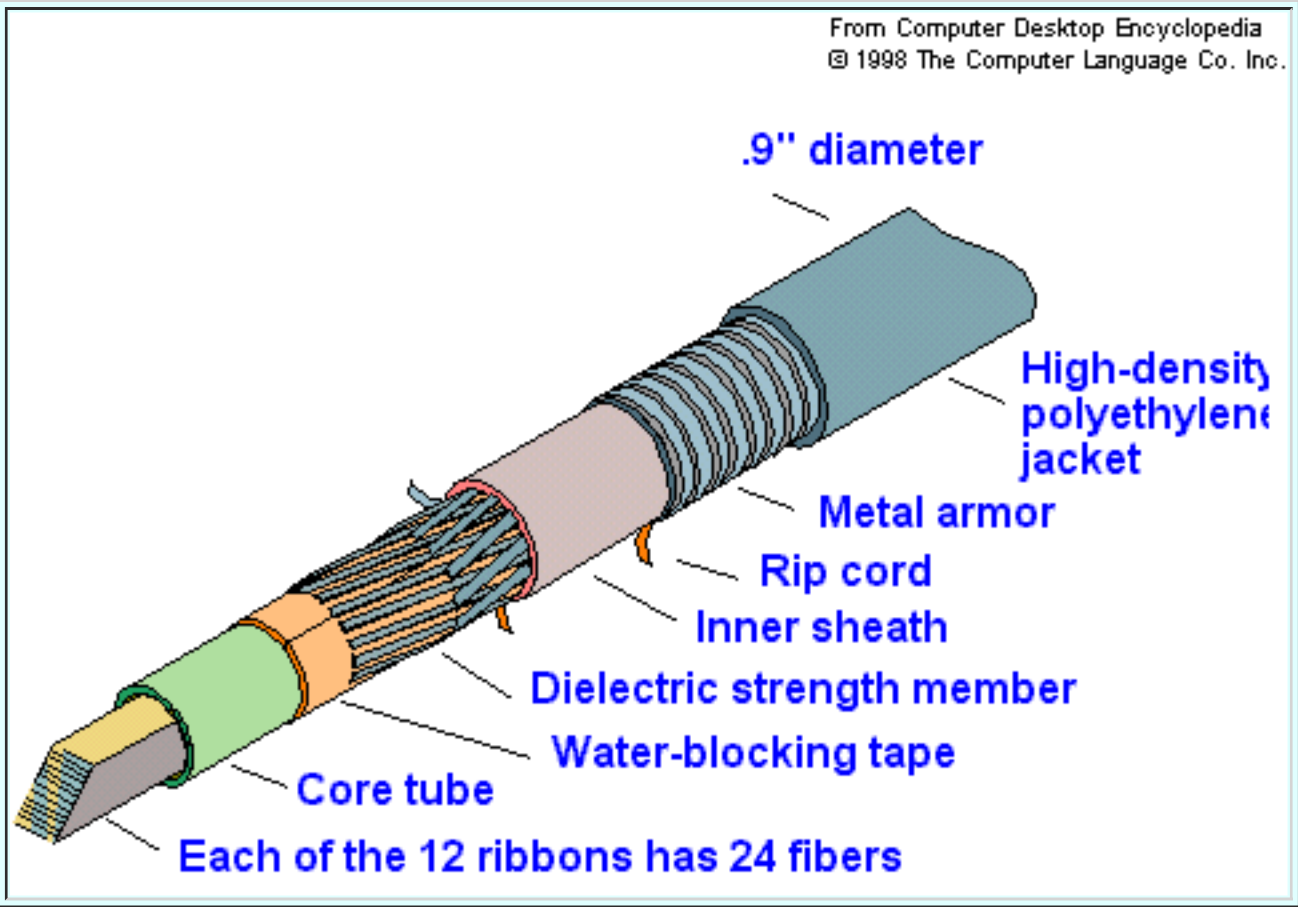
- *Commonly* used Transmission media
    - ***Commonly*** **used** wired **transmission media:**
        - **Twisted pair** of (**copper**) wires (good)

        - **Coaxial cable** - (better)

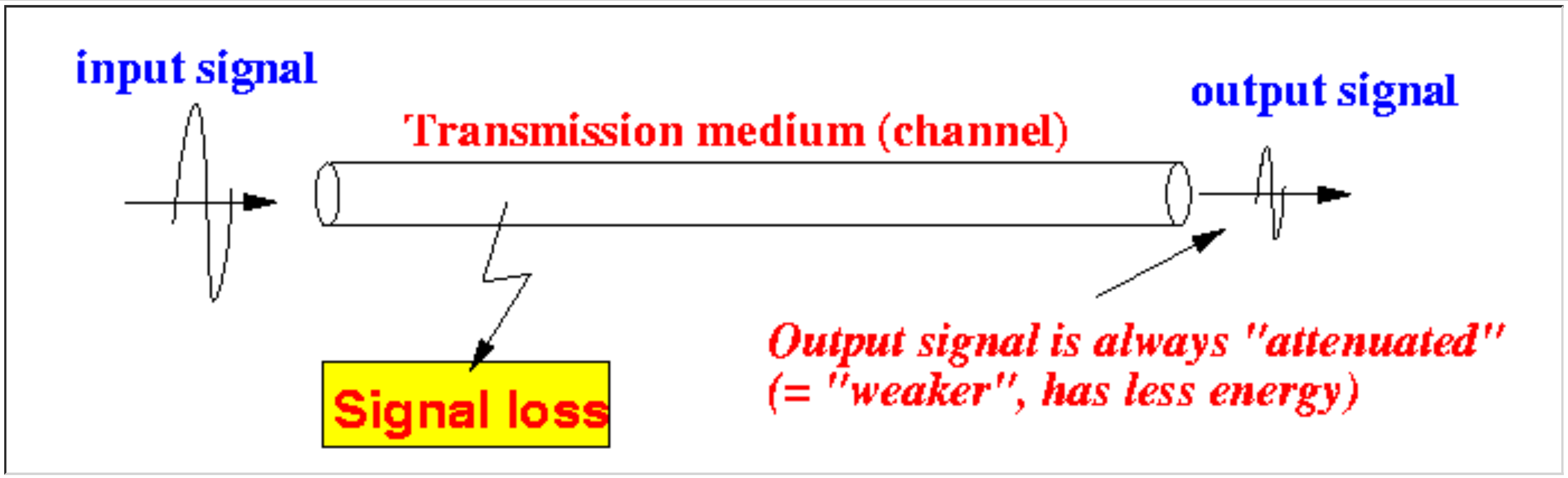
        - **Optical Fiber** (best)

Transmission over a copper wire

- **Energy loss in signal transmission in copper wires**
  - **Physics:**
    - When an **electrical signal (= electricity)** ***traverse*** through a **copper wire**, the **always lose** some ***power***

    Graphically:

    Terminology:
    - **Attenuation** = **loss** of **power**

- **Source of power loss**
  - There are **2 ways** that an **electrical signal** can **lose power**:
    - **Resistance** (in the **wire**):

      Resistance in the **wire** will cause ***some*** of the **energy** in the **signal** to be **converted** into ***heat***

    - **Radiation**:

When the **frequency** of the **electric current** is ***very high***, an ***alternating* current** will generate a **strong *electro-magnetic* field**.

The **wire** will work like an ***antenna*** - and some **energy** of the **signal** will ***radiate*** away....

- **Fact** from **Physics**:
  - The **amount** of **power loss** depends on:
    - The ***resistance*** of the **transmission medium**
    - The ***frequency*** of the **electro-magnetic signal** (= how **much** of the **power** will **radiate** away).

- **Energy loss characteristics**
  - The **energy loss *characteristics*** is **as follows:**
    - When the **(*alternating*) electric current** has a **very *low* frequency**:
      - the **current** encounters ***high* electrical *resistence***.

        (**DC** current generates a **low** of **heat** !!!!)

      Result:

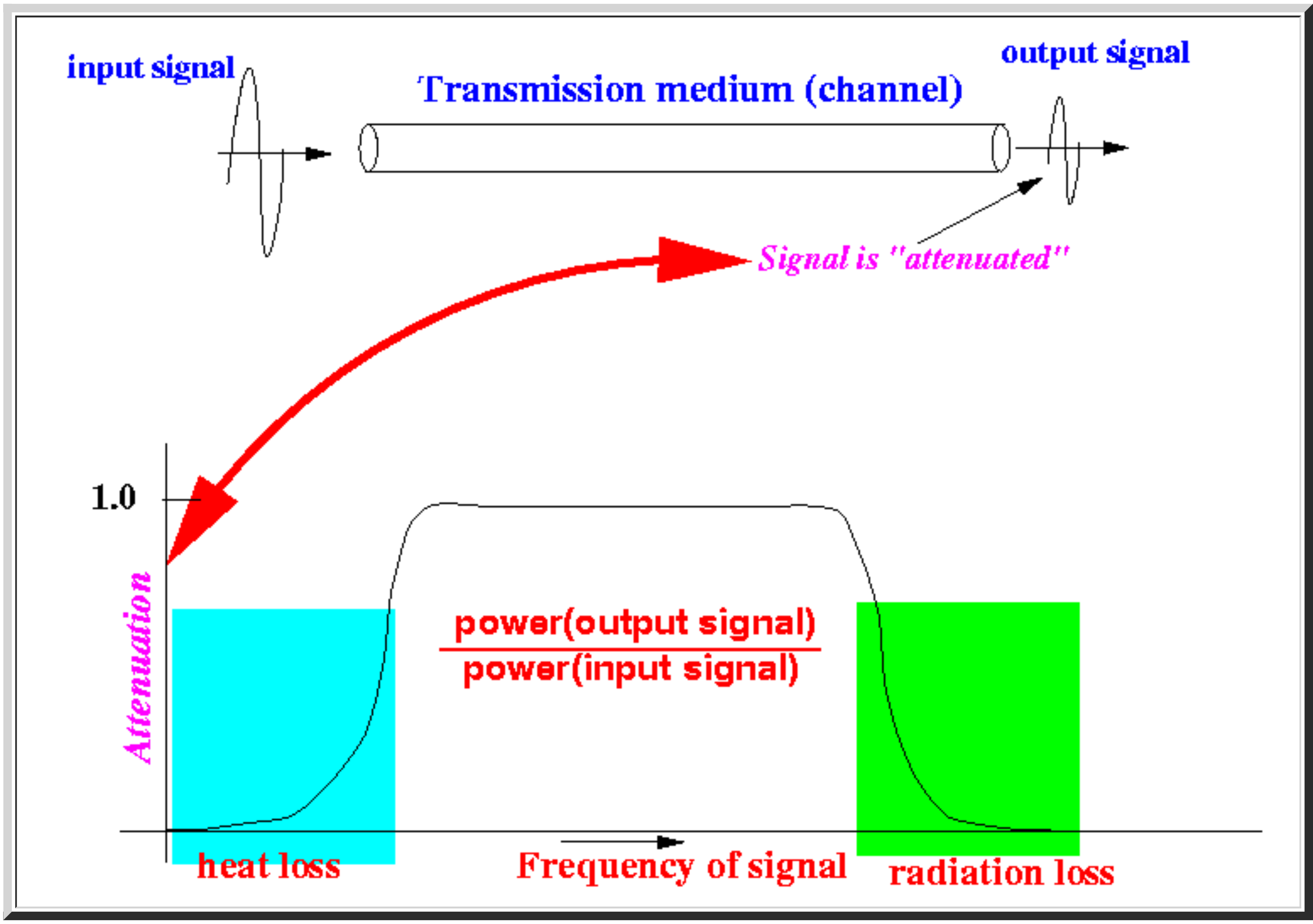
      - A **large portion of the energy** in the **transmitted signal** is **lost** because the **energy** is converted into ***heat***

    - When the **(*alternating*) electric current** has a **very *high* frequency**:
      - the **current** will **generate** a ***strong* electro-magnetic *field***.

      Result:

      - A **large portion of the energy** in the **transmitted signal** is **lost** because the **energy** is converted into ***electro-magnetic* wave (radiation)**

- **Energy loss *characteristic*** of a **wire** transmission medium is **as follows**:



The *bandwidth* of a transmission medium

- ***Acceptable* level of power *loss***
  - **Acceptable loss threshold** (set by **electrocal engineers**):

```
    power(output signal)
  ----------------------- ≥ ln 2 ~= 0.7
    power(input signal)
```

    This **value** is known as the **half-power point**: click here

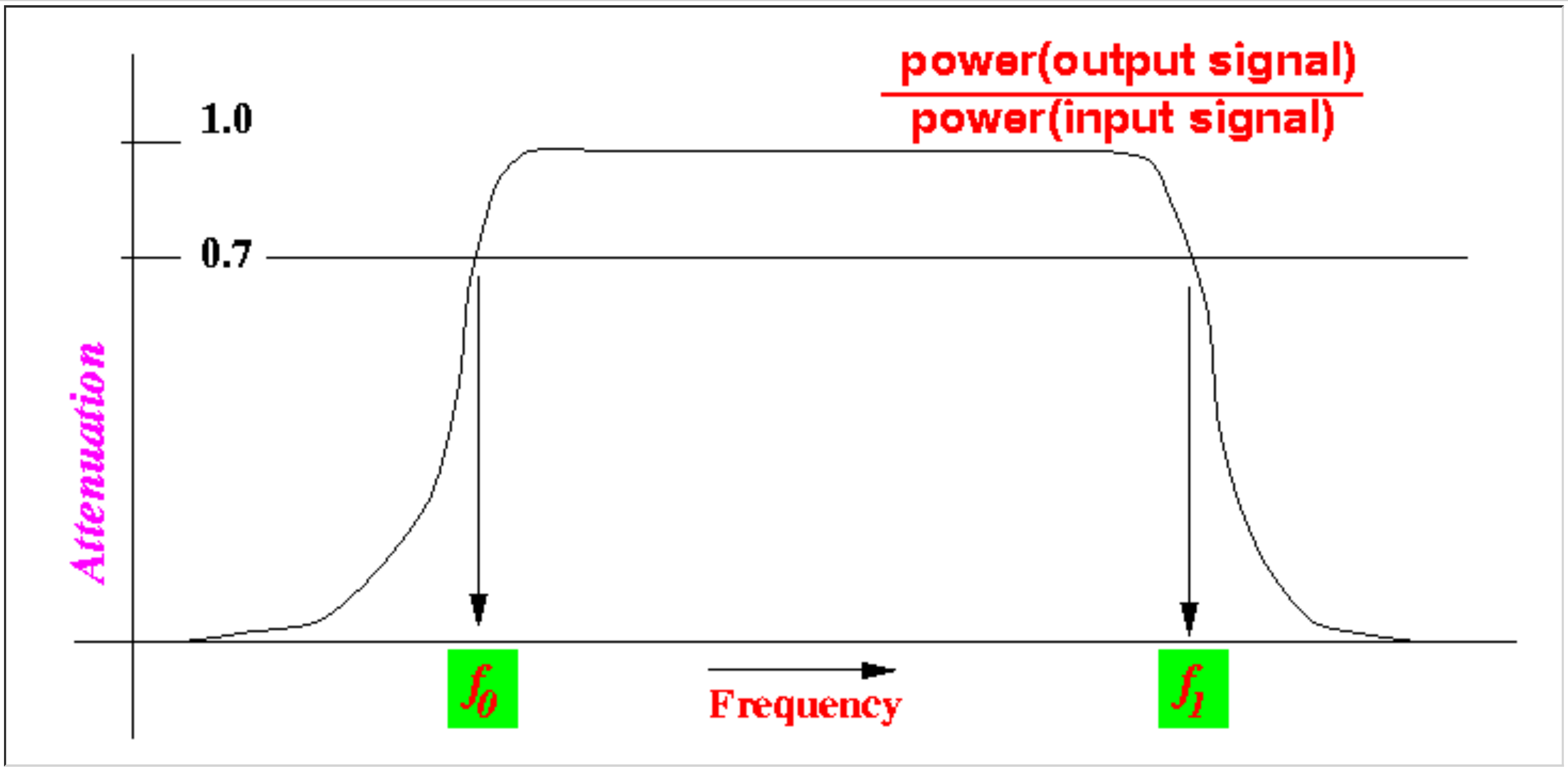
- **Range of *acceptable* frequencies of a transmission media**
  - **Definitions:**
    - Let f_0 = the **low end frequency** such that:

```
    power(output signal)
  ----------------------- = ln 2 = 0.7
    power(input signal)
```

    - Let f_1 = the **high end frequency** such that:

```
    power(output signal)
  ----------------------- = ln 2 = 0.7
    power(input signal)
```

  - **Graphically:** the ***frequencies*** f_0 and f_1 are defined as follows:



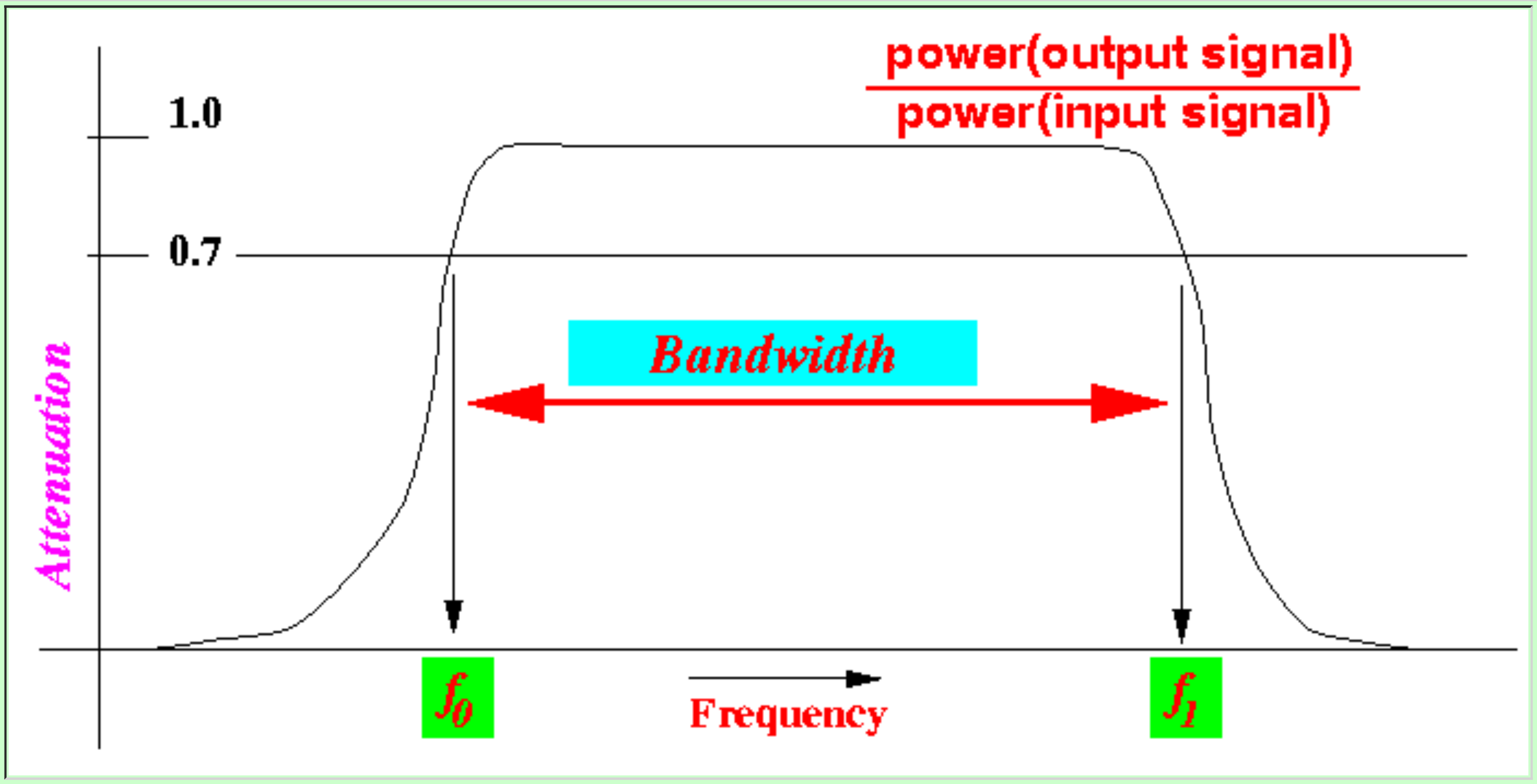
  - **Fact:**
    - **Electrical signals (= alternating current))** that have a **frequency** between f_0 and f_1 will pass through the

transmission medium with relatively *little loss* of energy

- ***Bandwidth* of a communication medium**
  - **Definition:** **bandwidth** of a **communication medium**:

    - **Bandwidth** of a **communication medium** (a.k.a., **transmission channel**) = $f_1 - f_0$

      Graphically:

    Fact:

    - There is a **direct relationship** between
      - **Bandwidth** of a **communication channel** **and**
      - **Maximum transmission rate** of the **communication channel** (= how ***fast*** you can **transmit** **communication channel**)

    This **relationship** is called the **Shannon formula** and will be discussed **next**

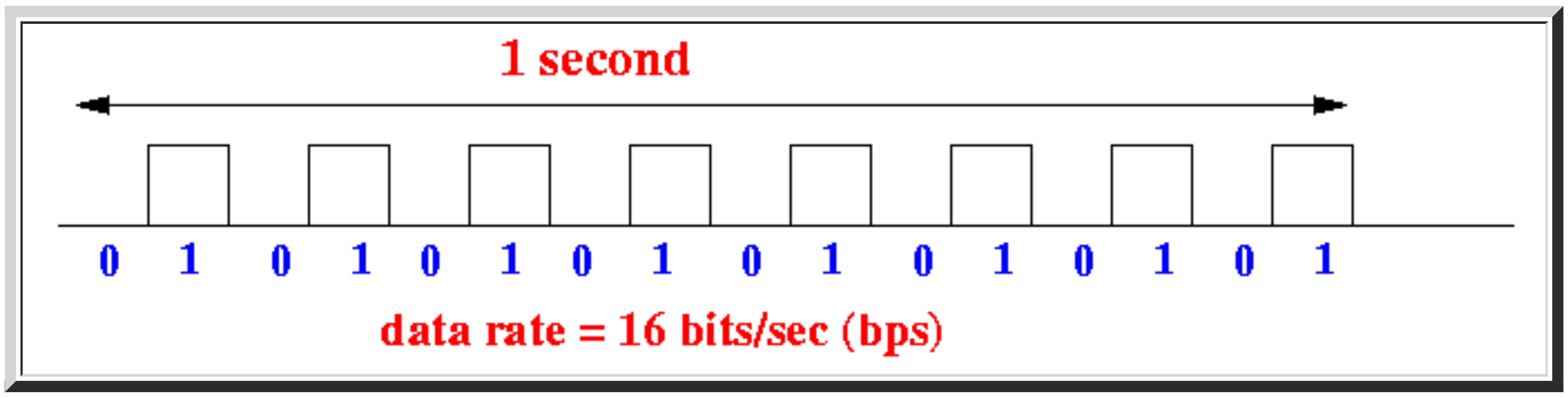
Data transmission rate and frequency

- **Data (transmission) rate**
  - **Fact:**
    - **Computer communication** often use ***digital* signalling**

      (I.e., computers transmit a **series** of **pulses** that represent **bits**)

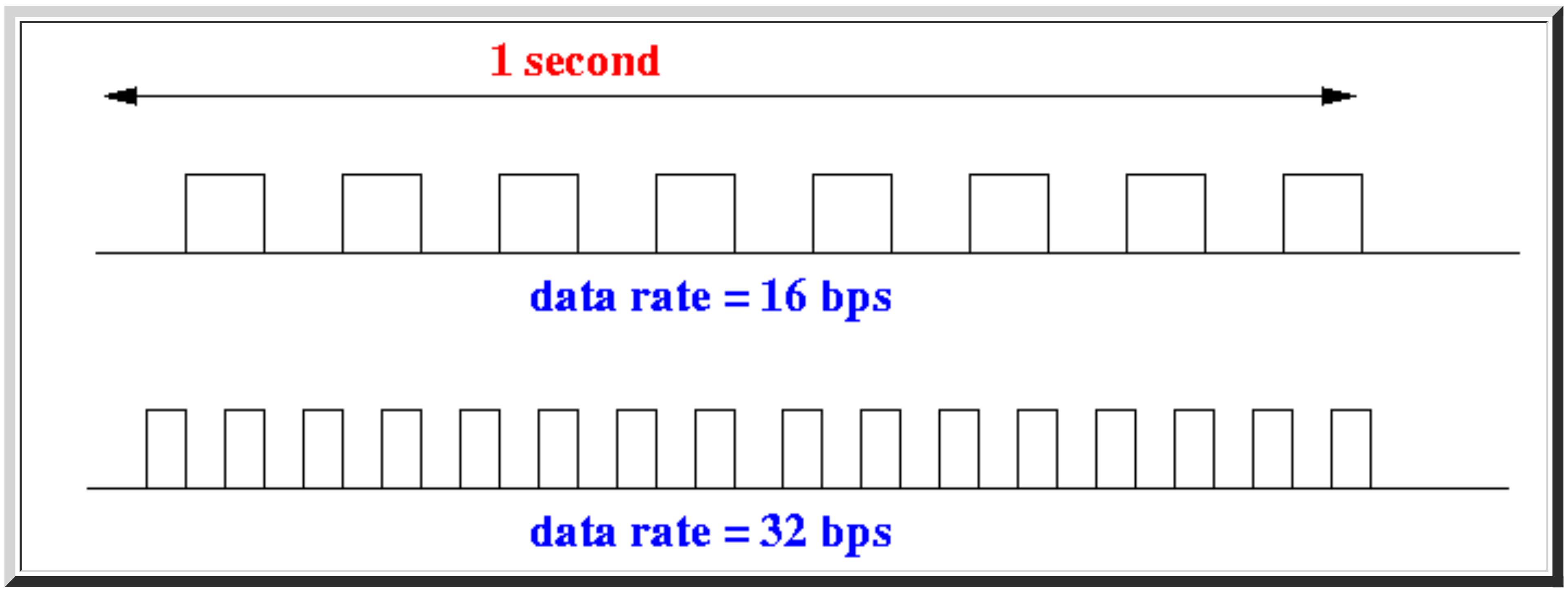
  - **Definition: data rate**
    - **data rate** (or data transmission rate) = **number of bits** transmitted in **one second**
    - **Unit** is: **bits/sec**

    Example:



- **Data rate and frequency**
  - **Observable fact:**
    - ***Faster* data rate** will result in a **signal** with a **higher *frequency***

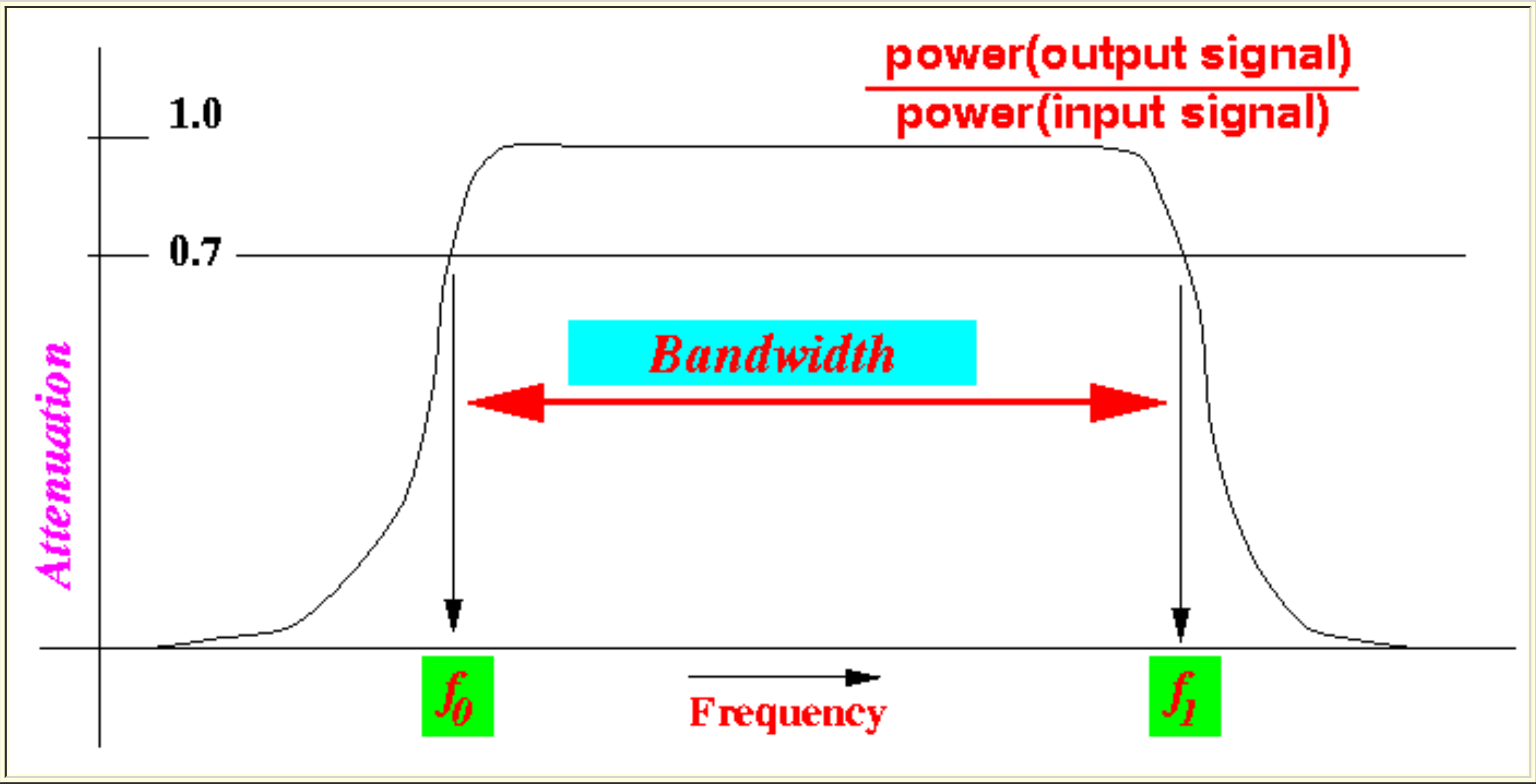
    Example:



    The **frequency** of the **second signal** is ***higher*** (more cycles per second)

- **Relationship between *bandwidth* and data transmission *rate***
  - **Clearly:**
    - A ***higher* data (transmission) rate** wil **result** in a **signal** with ***higher* frequencies**

- **Limitation**:
  - When the **frequency** of the **transmitted signal** is ***too* high**:
    - The ***signal*** become ***unrecognizable*** due to ***attentuation*** !!!



The *Maximum* data rate of a transmission medium

- **Another factor that affect reception quality**
  - **Fact:**

    - Beside ***physical* constraints** imposed by the **property** of the the **transmission media**, **another factor** that **affects** the **quality** of the **received signal** is:

      - ***noise***

- ***Maximum* data rate of a transmission channel**
  - The **factors** that **determine** the **data transmission rate** over a **transmission medium** (from electronics):

    - **Bandwith** of the **transmission medium** (i.e., the **physical property**)
    - **Ambient noise**

  - ***Theoretical* maximum data rate** of a **transmission channel:** (***no* proof**)

    - ***Maximum* data rate** of a **transmission medium** is:

      $$C = B \log_2 \left(1 + \frac{S}{N}\right)$$

      where:

```
C = maximum data rate (in # bits/sec)  --- a.k.a. capacity

B = Bandwidth of the transmission medium (in Hz)
S = power of the transmited signal (in Watt)
N = power of the ambient noise (in Watt)

S/N is called the "Signal to Noise ratio"
```

  Note:

- This **famous result** is derived by the late Mathematician **Claude Shannon** (see: **click here**)

Sharing a Transmission Medium

- **Motivation to *share* a communication medium**
  - **Fact:**
    - The **capacity (data rate)** of a **transmission medium** is ***very* large**

      Example:

      - ***One* optical fiber wire** can **transmit ≥ 1 TeraBits/sec** !!!

        (See: click here)

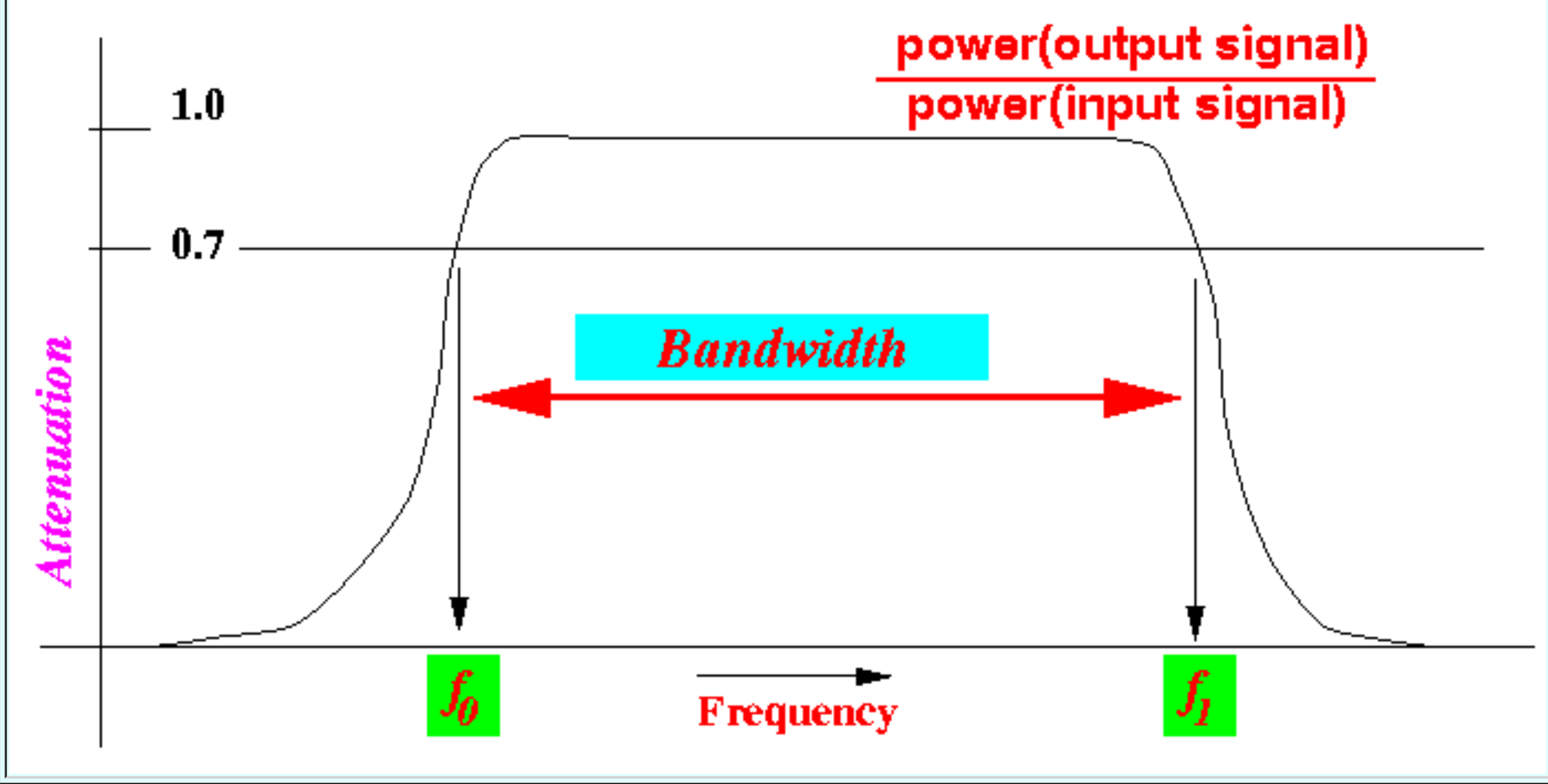
    - A ***single* user** can **not fully** use **1 TeraBits/sec** data rate...
  - **Common practice:**
    - ***Multiple* users** usually ***share*** one **single communication channel**

- **Terminology**
  - **Multiplexing:**
    - **multiplexing** = the **techniques** used to ***share*** a **communication medium**

- **Multiplexing Techniques**
  - There are **2 *commonly used* multiplexing techniques** used to **share transmission capacity** in the **physical layer** (= **signal transmission layer**):
    - The **transmission channel** can be **shared** in the ***time* dimension**:
      - ***Time* division multiplexing**
    - The **transmission channel** can be **shared** in the **"*space*" (frequencies) dimension**:
      - ***Frequency* division multiplexing**

      Note:

      - The **"space"** is a **communication medium** is the **range** of frequencies inside **bandwidth** of the **transmission medium**:

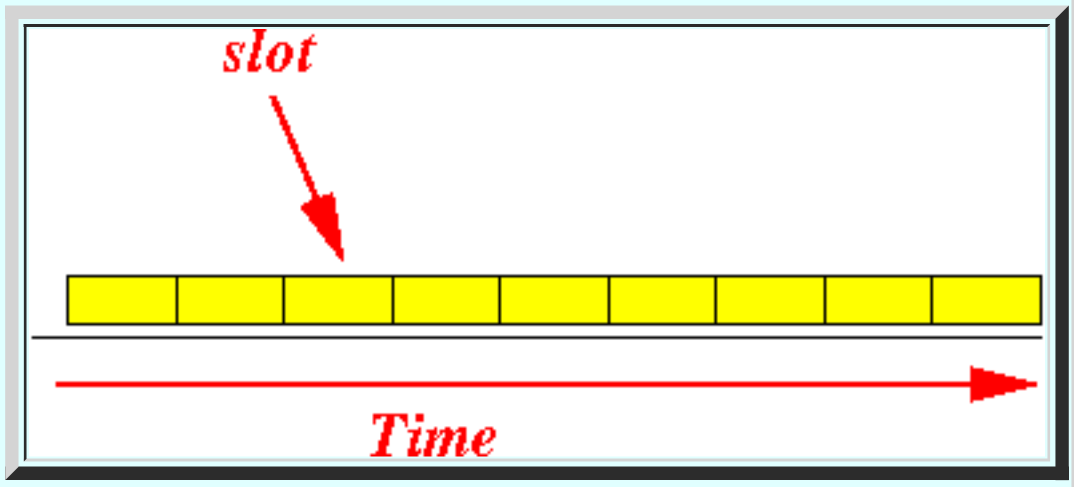


- **Analogy:**
  - **Sharing** a **house** with **4 rooms** among **4 persons**:
    - **Time sharing**:
      - **Each person** gets to **use** the ***entire*** **house** for **25% of the time**

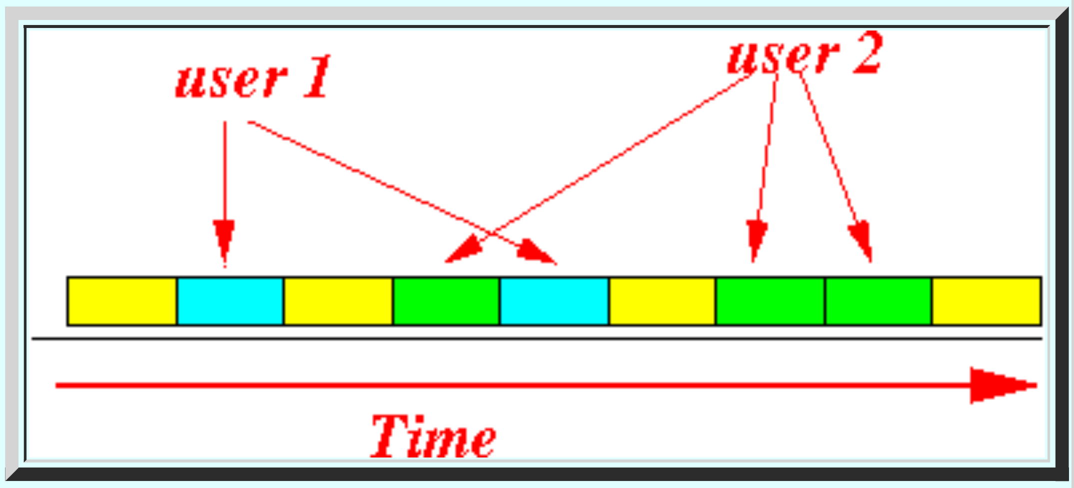
    - **Space sharing**:
      - **Each persn** gets to **use** ***one*** **room** **all the time**

Synchronous/Asynchronous Time Division Multiplexing

- **Sharing a channel in temporal dimension: Time Division Multiplexing (TDM)**
  - The **basic idea** of the ***Time* Division multiplexing** technique:
    - **Divide** the **time** into **slots**:

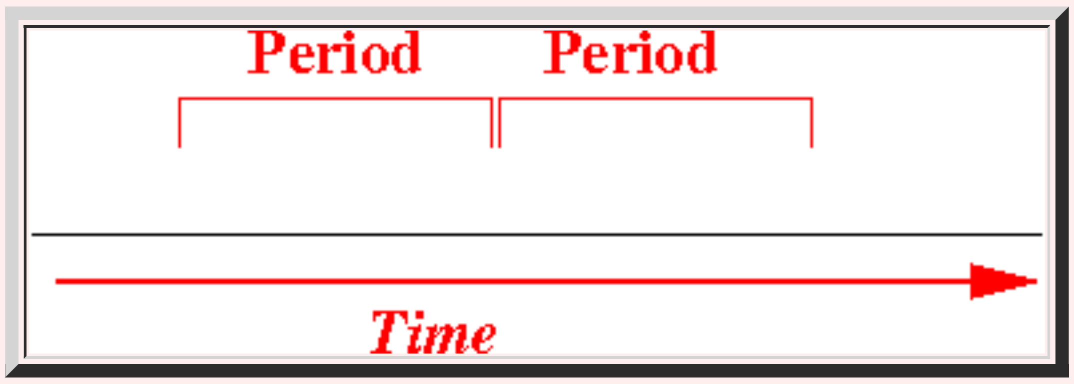
      

    - The **time slots** are **shared** among ***multiple* users**:

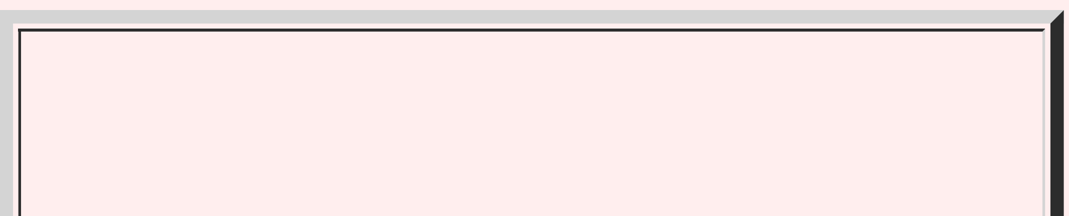
      

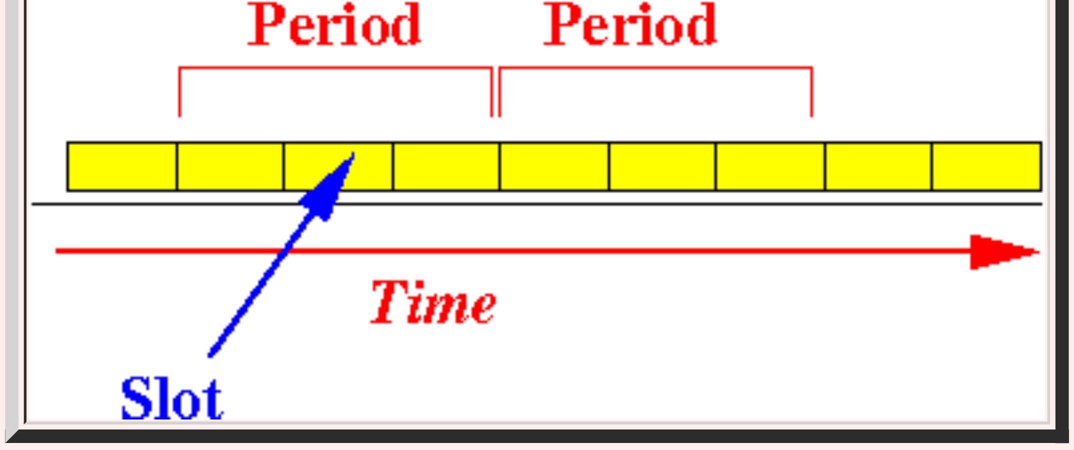
  - The **2 variants** of the **TDM** technique:
    - **Synchronous TDM**
    - **Asynchronous TDM**

- **Synchronous TDM**
  - The **Synchronous TDM** technique:
    1. **Time** is **divided** into **periods**:

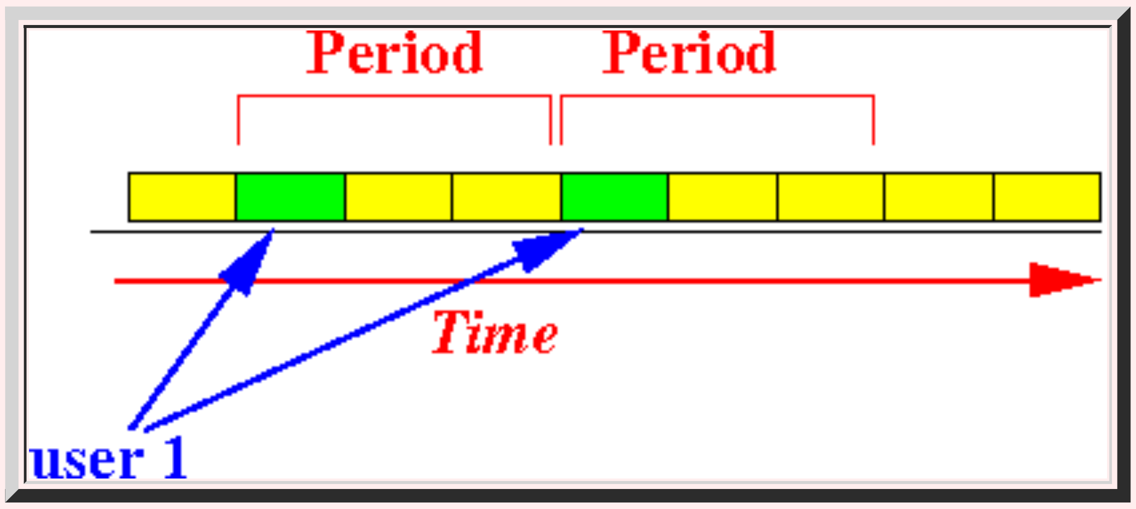
       

    2. Each **period** is **further divided** into **time slots**:

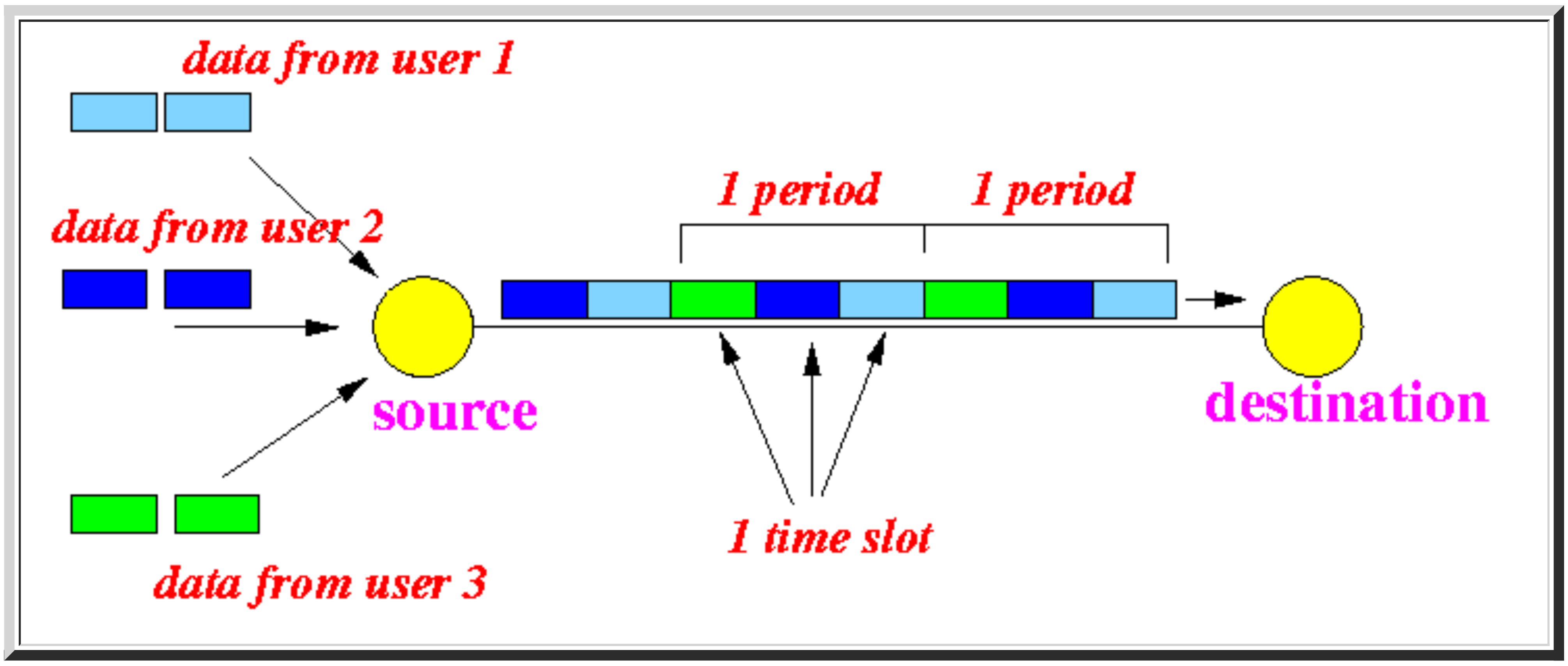


3. A **user** is **assigned** **one or more time slots** in **every period**:



The **slot** assigned is ***always*** in the **same position** inside a **period**

Example:

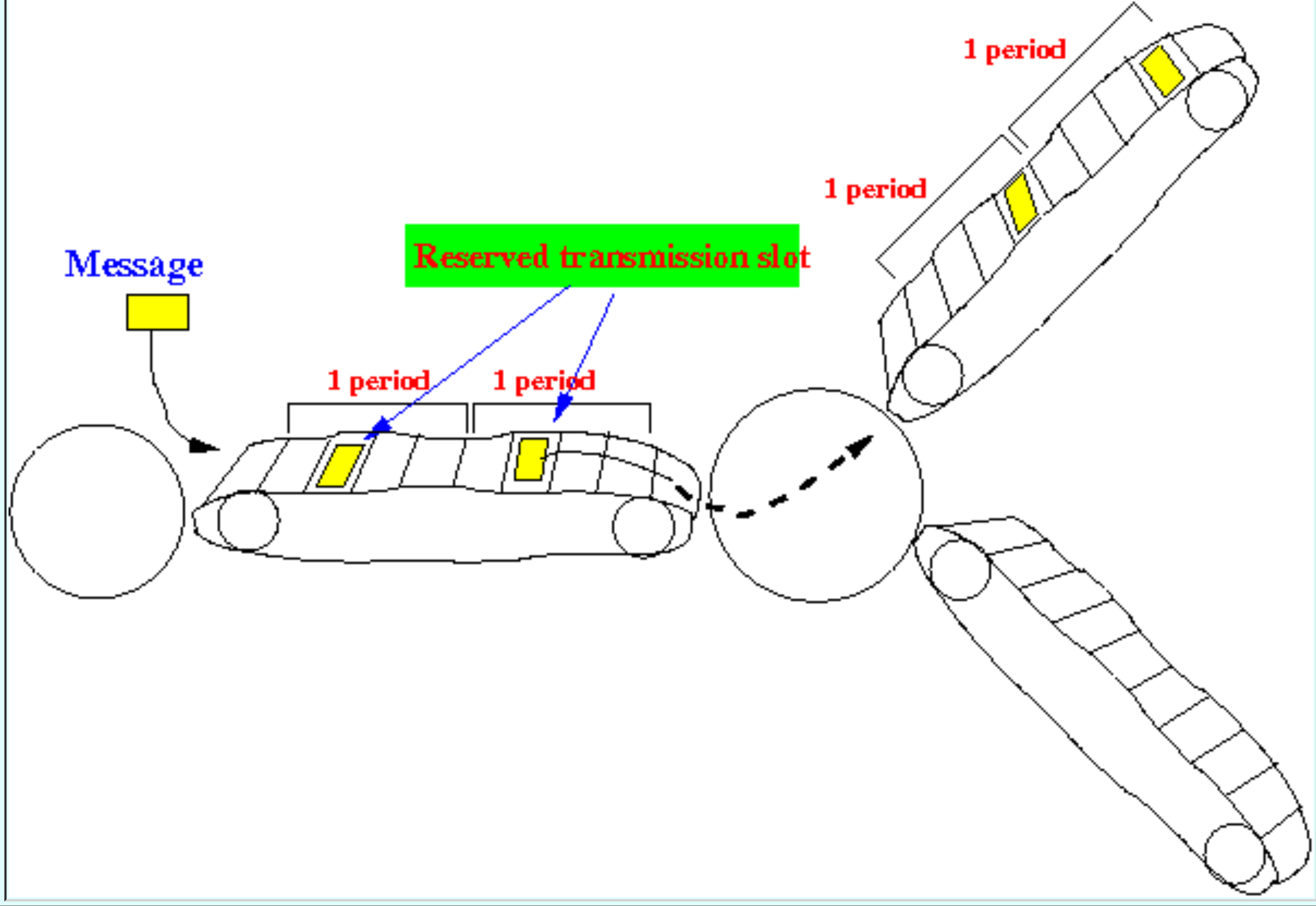


- **Advantage/disadvantage of *Synchronous* TDM**
  - **Advantage** of **Synchronous TDM**:
    - **Simple** (= **easy**) to **implement**

    - ***Synchronous* TDM** can provide ***performance guarantee***
      - ***Asynchronouus* TDM can** provide a ***fixed* waiting time guarantees**

        (The **waiting time** is ***at most* 1 period** !!!)

    Because the **Synch. TDM** method works like a **converyer belt**:



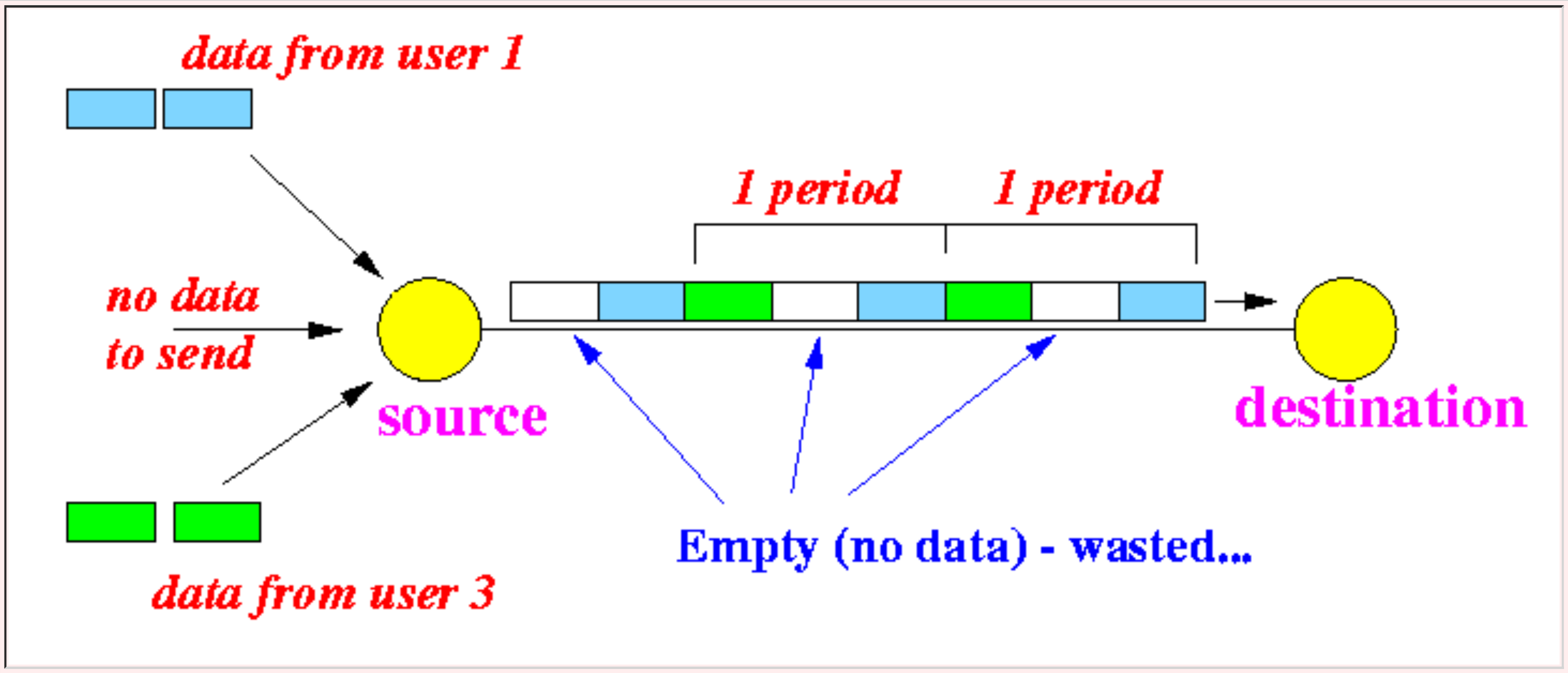
A ***waiting*** **message** will be **guaranteed** to be **transmitted** during the ***next*** **period** !!!

- **Disadvantage** of **Synchronous TDM**:

  - **Time slots** is ***wasted*** if the **user** has **no data** to transmit.

    - **Because** a **time slot** can ***only*** be used by that ***specific*** **user** !!!

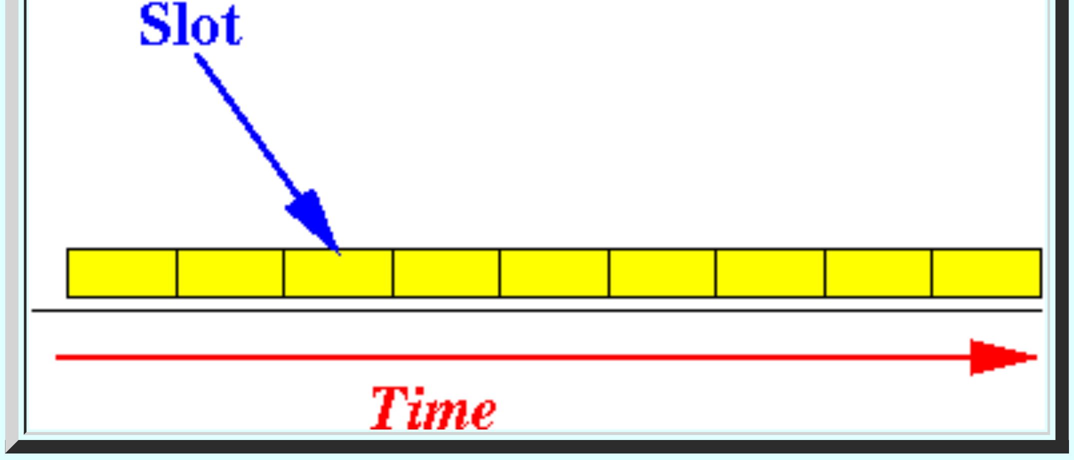
  Example: if **user 2 idle**, its slows are **wasted**



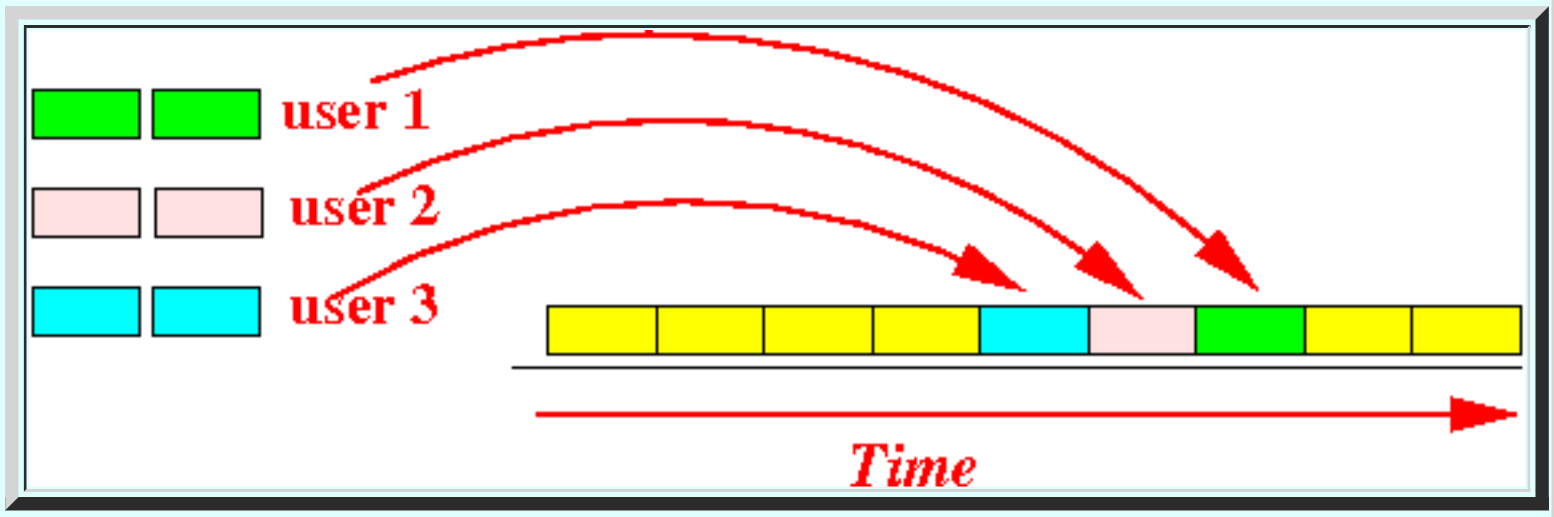
- **Asynchronous TDM**

  - The **Asynchronous TDM** technique:

    1. **Time** is **divided** into **slots**:



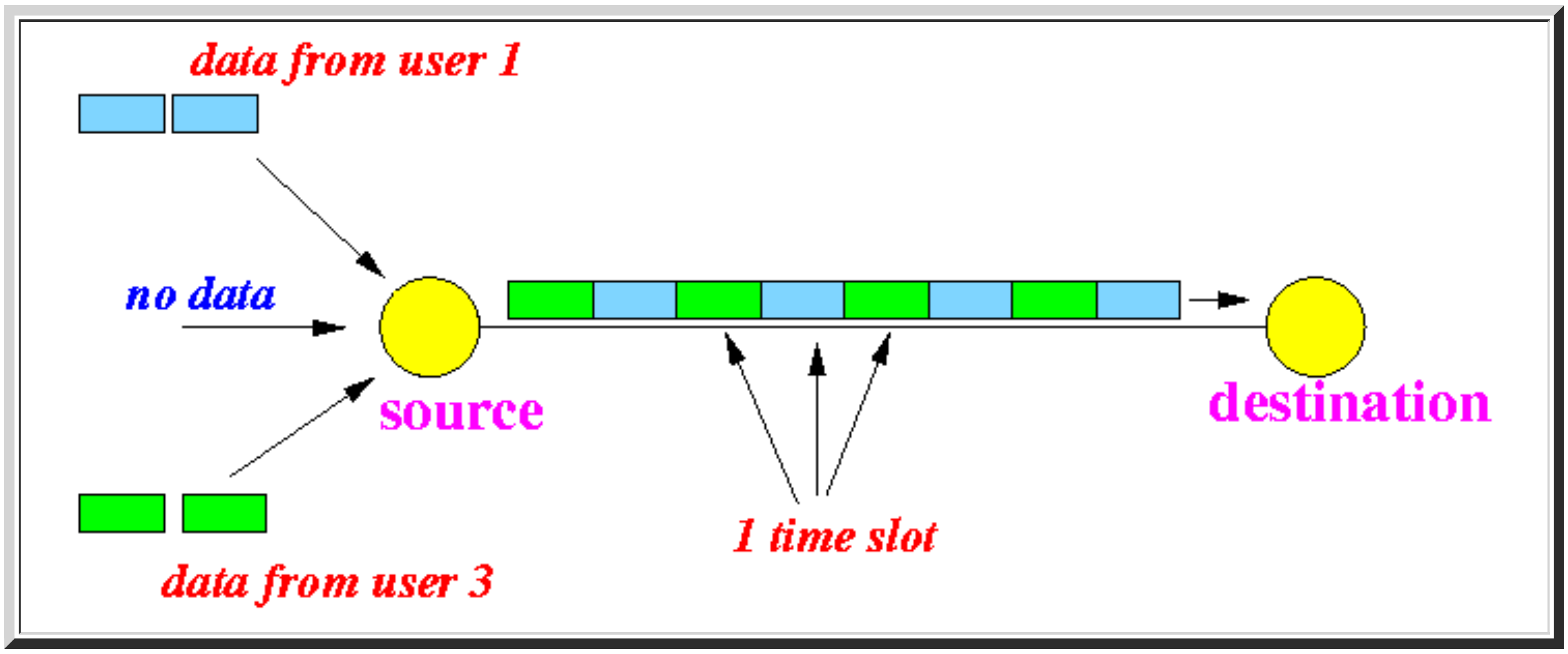
2. The **slots** are **shared** by **all** ***active* users**:



Note:

- A **slot** will ***only*** be **wasted**, if ***all*** **users** do ***not*** **have data** to transmit,

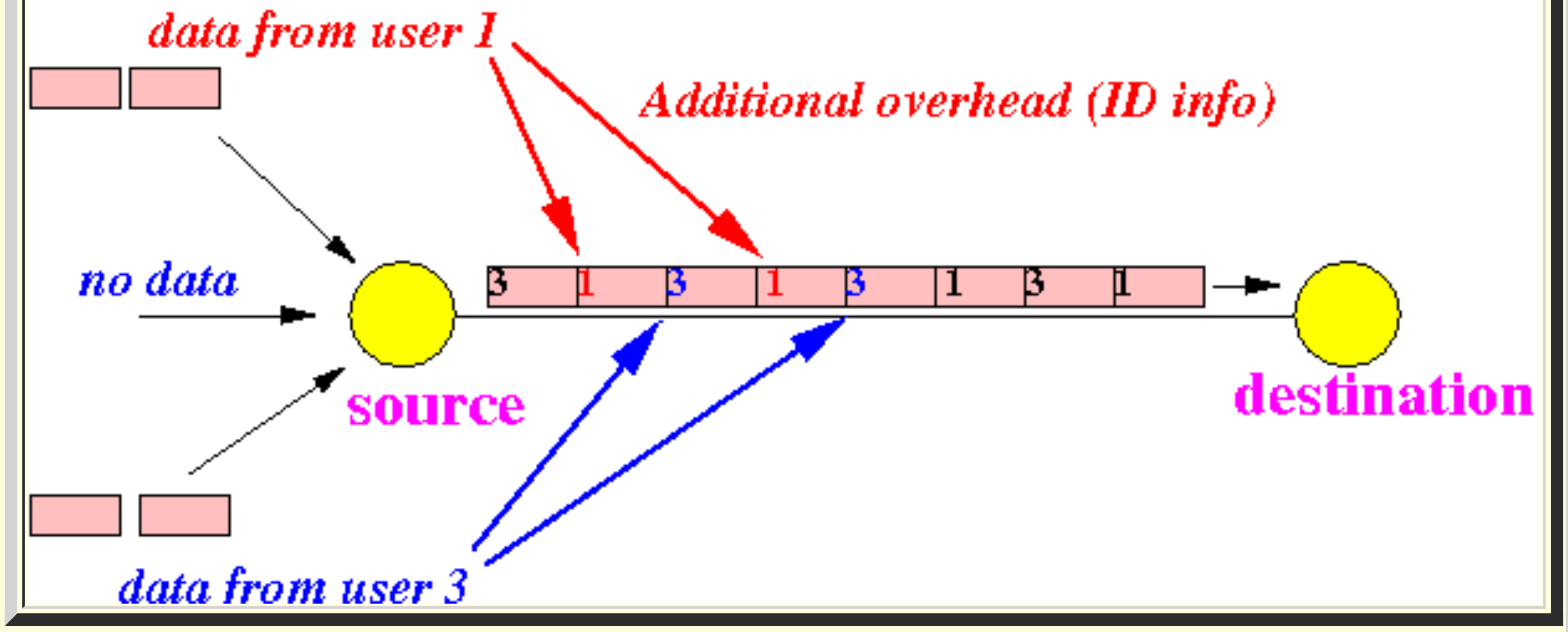
Example:



- **Implementation note:**

  - I used **colors** to indicate the **sender** of the **packets**....

  - In reality, the **packet** are **distinguished** by a **source/destination ID**:



- **Advantages/disadvantages of *Asynchronuous* TDM**
  - **Advantage** of ***Asynchronuous*** **TDM:**
    - ***Efficient*** **use** of the **transmission capacity**.
      - **Asynchronuous TDM** will ***only*** **waste transmission capacity** (= **slots**) when ***all*** **senders** are **idle**

  - **Disadvantage** of ***Asynchronuous*** **TDM:**
    - It's **much *harder*** to provide **service guarantees because:**
      - The **time** that a **user** gets a **transmission slot** depends on the ***current*** **number** of **active users**
      - ***Asynchronuous*** **TDM** can **not** provide a ***fixed*** **waiting time guarantee**

    Note:

      - ***Asynchronous*** **TDM** can provide a ***soft*** **guarantee** on **delay (waiting time)** using ***flow control***

        Flow control = **controlling** the ***transmission rate*** of **data** over a **communication link**

Frequency Division Multiplexing

- **Sharing a channel by "space": Frequency Division Multiplexing**
    - ***Frequency* Division multiplexing**:
        - The **range (space) of frequencies** is **divided** among the **users**
    - **Fact** from **Physiscs**:
        - The **signals** sent using ***different*** **frequency ranges** do ***not*** **interfere** with each other
    - **Example:**
        - **Radio**
            - Transmissions on **FM 98.5** do ***not*** **interfere** with transmissions on **FM 94.1**....

- **Problem with *Frequency* Division Multiplexing**
    - **Problem:**
        - **Users** that use ***different*** **frequencies** to **transmit** can ***not*** receive ***each others'*** **transmissions** !!!

    Result:

        - **Freq. Div. Multiplexing** is ***not*** used in **computer communication** !!!
    - **Note:**
        - ***Frequency*** **division multiplexing** is ***mainly*** used in **radio** or **walkie-talkie** communication....

Intro: Error *Detection* and *Correction* at the Physical Layer

- **Errors at the Physcial Layer:**
  - **Fact:**
    - A **transmitted signal** can be **received** ***incorrectly***

      Example:

      - A **"1" signal can** be **received** (and **decoded**) as a **"0" signal**
  - **Common causes** of **transmission errors:**
    - ***Background* noise**:
      - **Radiation** (from space) can cause ***single* bit errors** in a **stream of transmitted bits**
    - **Weather related** (i.e., **lightning**):
      - **Lightning** can cause ***multiple consecutive* bit errors** in a **stream of transmitted bits**

- **Typical error rates**
  - **Typical error rates**:
    - **Error rate** in **twisted pairs (copper wires)**:
      - **1 bit error** in every few **1000 bits** transmitted.
    - **Error rate** in **optical fibers**:

- **1 bit error** in several **1,000,000 bits** transmitted

- ***Why* detect errors in the *Physical* layer**
  - **Detecting** transmission **errors**:
    - **Error detection** is a **part** of the function to ***provide* reliable communication**

    - The **layer** (see **OSI model**: click here) that is **responsible** for providing **reliable communication** is:
      - The ***data link* layer**

        (And ***not*** the ***physical* layer**)

    - **Technically**, it is ***not* necessary** to **detect errors** at the **Physical Layer**:

  - ***Why*** detect errors at the ***physical* layer**:
    - ***Early* error detection** in the **Physical layer** will result in **faster *response***

    - It is **also** more **efficient**:
      - The **sooner** we detect an **error** (and ***stop*** the **processing** of a **message**), the **less computation resources** we will **waste**

Principle of error *detection* and *correction*

- **Illustrative *question***
  - **Example** of ***transmitted*** **data**:

    ```
    11111111
    ```

  - **Example** of ***received*** **data**:

    ```
    11110111
    ```

  Question:

  - **How** can we **tell *whether some* bit(s)** is/are ***incorrect*** (***without*** know **what** the **transmitted data** was) ???

- **Principle of error *detection/correction* methods**
  - **Principe** to **detect/correct** of **errors**:

    1. **Embed *extra* bits** into the **transmitted message**
       - The ***insert*** **extra bit** creates a **certain *pattern*** in the **bits stream**

         (usually conforming to some **(mathematical) property**)

    2. The **receiver checks** for the **embedded *pattern*** in the ***received* data**:
       - **Pattern** was **found** $\Rightarrow$ **message** assumed to be **correct**
       - **Pattern** was ***not* found** $\Rightarrow$ **message** assumed to be in ***error***

- **Mathematics: *coding theory***

- **FYI:**

  - The **mathematical theory** that deal with **design patterns** in the transmission is called **coding theory**.

  I will only discuss some **commonly used codes**

- **Commonly used error detection/correction schemes**

  - **Commonly used schemes:**

    - **Error *detection* schemes:**

      - **Parity checks**
      - **Cyclic Redundancy Check** (CRC)

    - **Error *Correction* Schemes:**

      - **Hamming's code** (can **correct 1 bit error**)
      - **Solomon-Reed's code** (can **correct 2 bit errors**)

One-dimensional parity-based error detection schemes

- ***Parity***
  - **Parity**:
    - **Parity** = the **state** of **being equal** (from **dictionary**)

  - **Parity-based** error **detection** scheme:
    - ***Add*** some **bits** to the **transmitted message** so that:
      - The **number** of **1 bits** in the **message** is ***even*** (or ***odd***)

    (**That** is an **exmaple** of a ***pattern*** that I was **eluding** to in a **previous webpage**....)

  - **Error *detection*** using a **parity based** scheme:
    - When a **message** is **received**, the **reciever checks** for:
      - the **parity property**

      to **verify** if there was an **error**.

- ***One-dimensional*** **Parity Scheme**
  - **Even** and **Odd Parity:**
    - **Even parity:**
      - **Add one extra bit** to the **message** so that the **total number** of **1's in the message** is **always *even***
    - **Odd parity:**
      - **Add one extra bit** to the **message** so that the **total number** of **1's in the message** is **always *odd***

  - **Examples:**

```
Transmitted data (unprotected):   1111000     1010101     1111111
```

```
Even parity:                        11110000    10101010    11111111

Odd parity:                         11110001    10101011    11111110
```

- **Error Checking:**

  - When the **received message** does not have an **even** (or **odd** **number of 1's** in the **even (or odd) parity method**, the **receiver** will **assume** that the **message** is **corrupted**

- **Property of the *one-dimensional* parity scheme**

  - **Properties:**

    - **Parity checks** can **detect** all ***odd*** **number** of **bit errors**

      - **Each error** will **change** the **number** of **1 bits**:

        - by **+1**:

          ```
          Before:   11100111      (6   1's)
          After:    11101111      (7   1's)
          ```

          or

        - **−1**

          ```
          Before:   11100111      (6   1's)
          After:    11100011      (5   1's)
          ```

      - An ***odd*** **number** of **bit errors** will **result** in:

        - A ***change*** of **parity**

          (I.e.: **odd parity** will become ***even*** **parity** and **vice versa** !!!)

    Example:

    ```
    Original data (unprotected):      1111000

    Even parity:                      11110000

    Data with 1 bit error:            11010000

      ===>  1 bit error results in odd parity !!!
             (error detected !!)
    ```

- **Parity checks** can **not** detect an ***even*** **number** of **bit errors**

  Example:

```
   Original data (unprotected):       1111000

   Even parity:                       11110000

   Data with 2 bit errors:            11010100

     ===>  2 bit errors results in even parity !!!

           Error cannot be detected !!!
```

Multi-dimensional parity-based error detection schemes

- **Two-dimensional parity**
  - ***Two-dimensional*** **Parity scheme**
    - ***2-dimensional*** **parity scheme**:
      - Form a ***MxN*** **matrix** of **bits**
      - **Add** a **(even or odd) parity bit** to **each *row*** **and** to **each *column***
  - **Example:**

```
Original data (unprotected):   1111000   1010101   1111111


1. For a matrix of bits:

                              1111000
                              1010101
                              1111111

2. Add parity bits:

                              11110000
                              10101010
                              11111111
                              10100101
```

  - **Properties** of the ***2-dimensional*** **parity** scheme:
    - The **2-dimensional parity scheme** can ***correct*** all **1 bit errors**

      Example:

```
Transmitted data:

                  11110000
                  10101010
                  11111111
                  10100101

Received with one bit in error:
```

```
            11110000
            10101010
            11011111    <---- odd parity
            10100101
               ^
               |
        odd parity
```

The **receiver** can **tell *which* bit** was in **error** from the **parity check** !!!

Therefore:

- The **receiver** can **take the initiative** to **correct** the **received message** !

- The **2-dimensional parity scheme** can **detect** all **2 bit errors**...

  but it ***cannot*** **correct** the error.

  Example:

```
   Transmitted data:

            11110000
            10101010
            11111111
            10100101

   ------------------------------------------

   Received with 2 bits in error:

            11110000
            10111010    <---- odd parity
            11011111    <---- odd parity
            10100101
               ^^
               ||
        odd parity
```

The **errors** can be **detected**.

However, the **receiver** ***cannot*** **correct** the error:

- The **cause** of error is ***ambiguous***:

```
   Original data:
```

```
                 11110000
                 10011010
                 11111111
                 10100101
```

```
Error pattern #1:

                 11110000
                 10001010    <---- odd parity
                 11011111    <---- odd parity
                 10100101
                   ^^
                   ||
             odd parity


Error pattern #2:

                 11110000
                 10111010    <---- odd parity
                 11101111    <---- odd parity
                 10100101
                   ^^
                   ||
             odd parity
```

Both cases will **result** in the ***same*** **parity pattern** !!!

The **receiver** **cannot tell** **which** of these **2 error cases** has **occured**....

So the **receiver** can **not** correct the **error**

Intro to Error *Correcting* Codes

- **Prelude: Coding Theory**

  - **Coding Theory:**

    - **Coding Theory** = a **discipline** of **Mathematics** that study the **properties** of **codes** and their fitness for specific applications.

  - **Use** of **codes**:

    - **Data *compression***
    - **Data *encryption***
    - **Error correction**

    Wikipedia: click here

- **Error correction codes**

  - **Key** to **error correction**:

    - ***Redundancy***....

    Example: a ***naive*** **error correction** code

    - For **every bit** that you send, **repeat it *3 times***

      Example:

      ```
      Message:          1011

      Transmitted as:   111000111111
      ```

      The **receiver** use **"majority voting"** on **groups of 3 bits** to decode the **message**

  - ***More*** **sophisticated** **error correction codes**:

    - ***Hamming* code**:

- Can **correct** ***single* bit errors**

- ***Reed-Solomon* code**:
  - Can **correct** ***2* bits errors**

I will ***only*** show you ***how to*** use ***Hamming* code**

No proofs.... (beyond the scope of this course !)

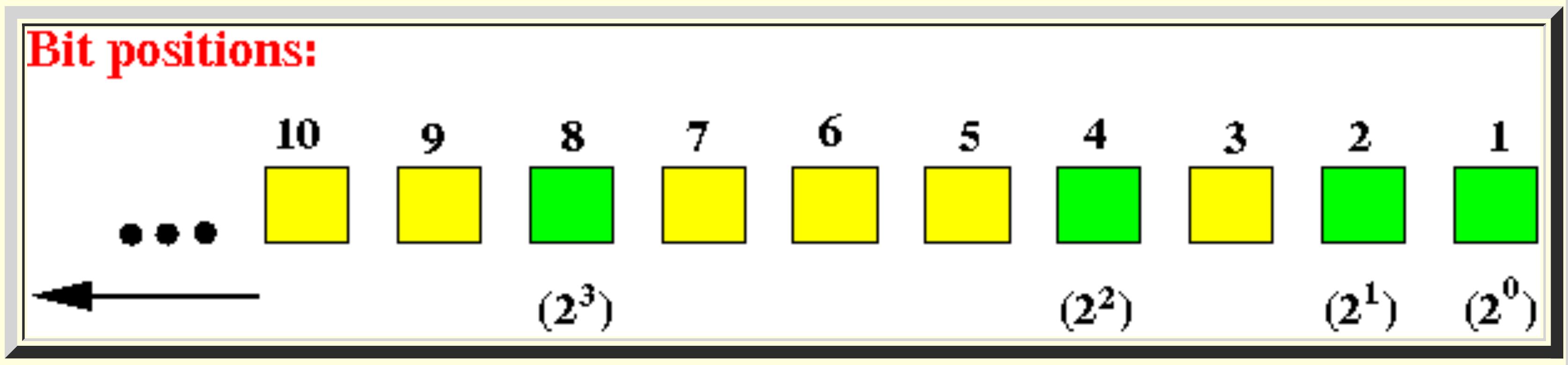
Hamming code: encoding procedure

- **The Hamming Code**
  - **Hamming Code:**
    - **Hamming code** is a **linear error-correcting code** named after its inventor, **Richard Hamming**.
    - The **Hamming code** can:
      - **Detect *and* correct** all **1 bit errors**
      - **Detect *most*** 2 bit errors --- but does **not** detect **all** 2 bit errors

      (It uses a ***minimum* number** of **extra bits** to achieve these properties)

- **The *encoding* procedure of the Hamming code**
  - **How to encode** data using **Hamming code**:
    1. **Input** = a **sequence** of **bits (0/1 values)**
    2. **Determine** the **number of *extra* bits** to insert (see **example**)
       Insert the **extra bit *positions*** into the **input bits**
    3. **Compute** the **values** of the **extra bit *positions***
    4. **Distribute** the **computed values** into the **extra bit *positions***

  - The **Hamming encoding procedure** illustrated by an **example**:
    1. **Input** = `101010`
    2. **Determine** the **number of *extra* bits** to insert:
       - The **positions** that corresponds to a ***power* of 2** are ***Hamming* check bit positions**:

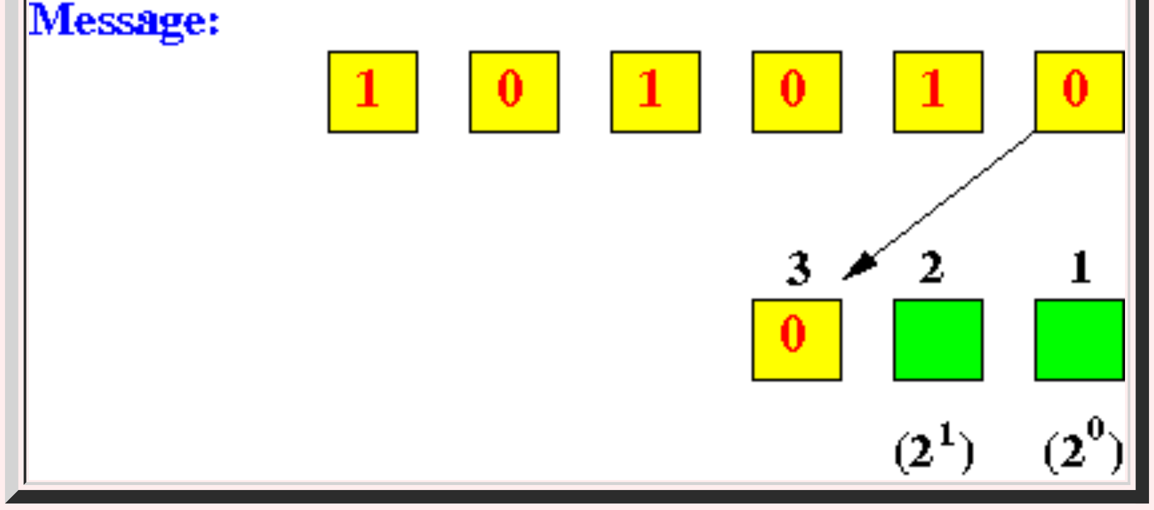


       - ***Add* extra bit positions** at **locations**:
         - 2^0, 2^1, 2^2, ... (and so on)

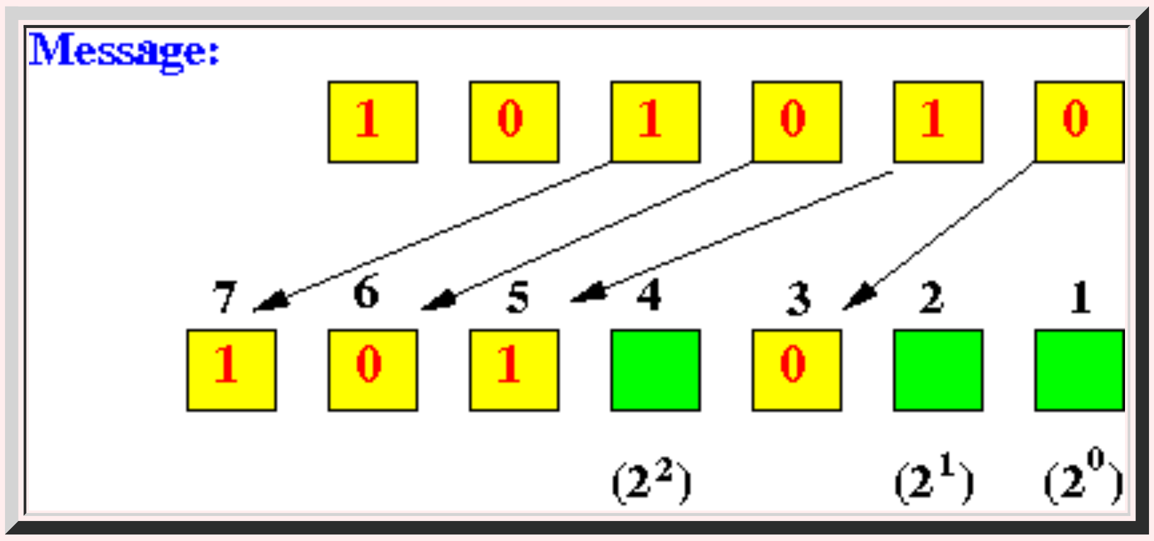
         until ***all* data bits** can be **accommodated**

       Example: input = **101010**

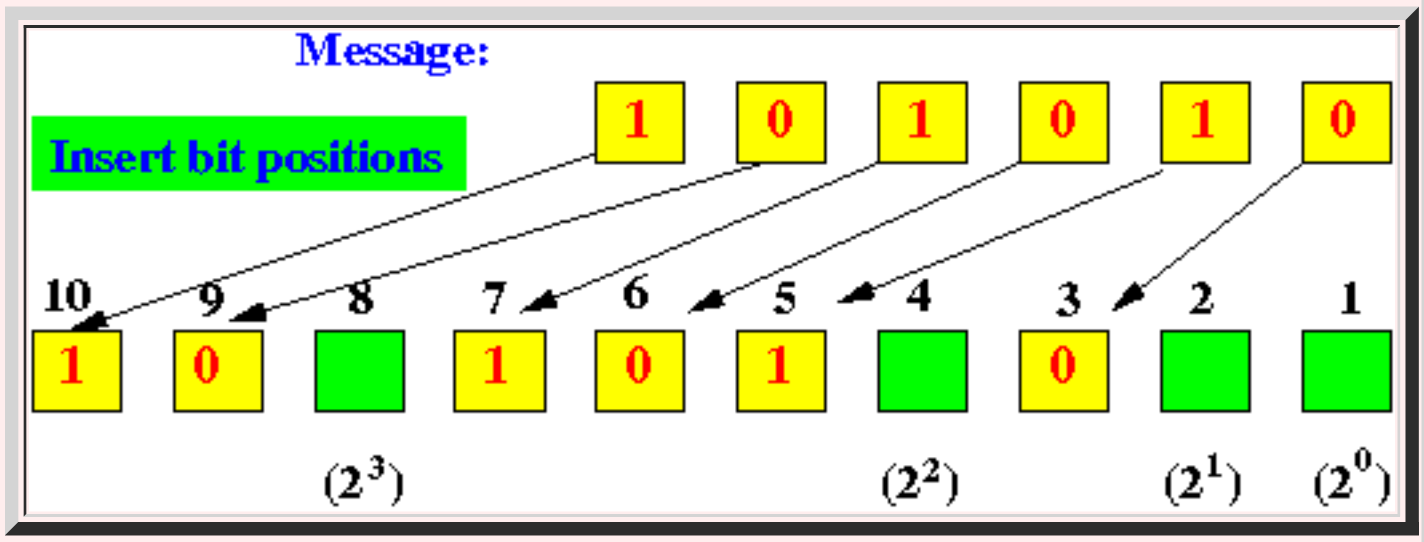
       - We start with:



- We need to **add** one more ***Hamming* check bit**:



- We ***still*** need to **add** one more ***Hamming* check bit**:



Now we are **done** !

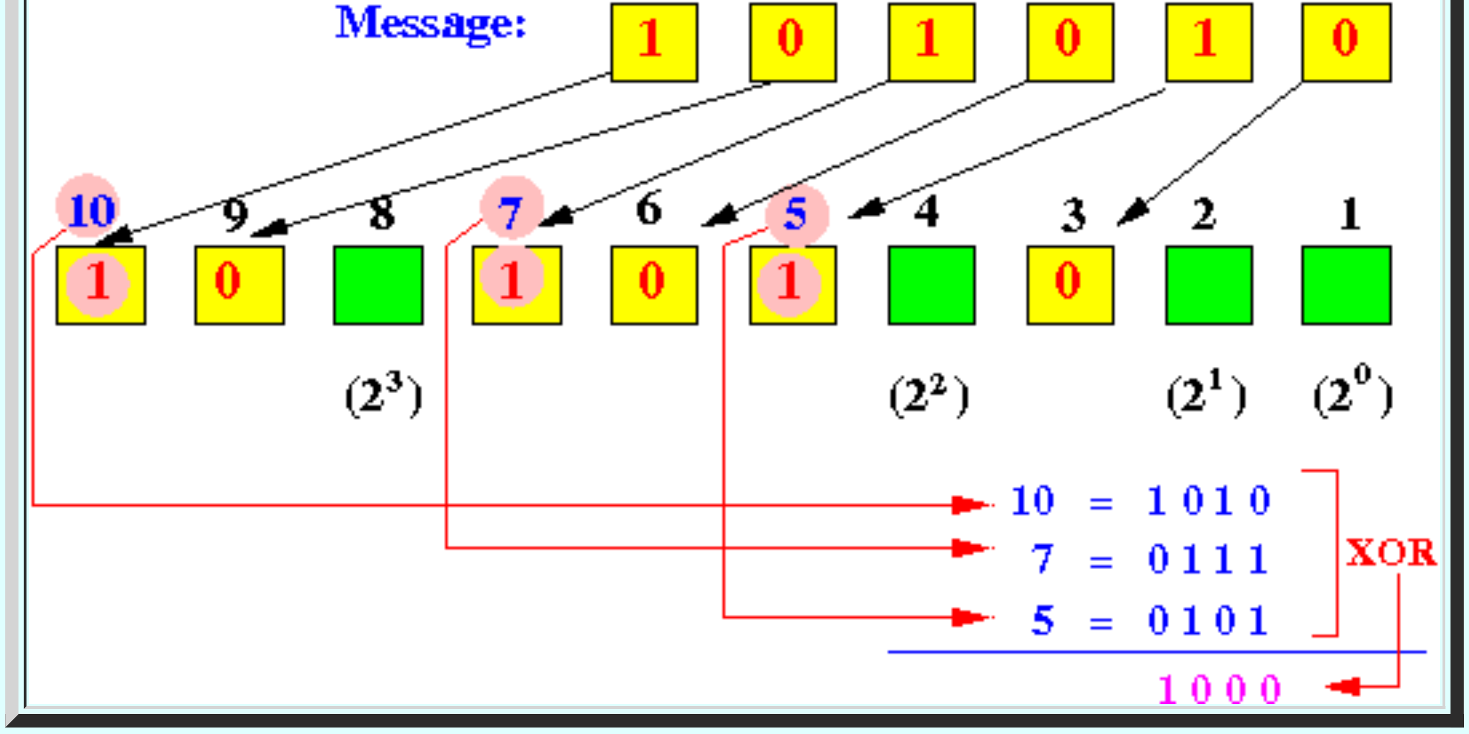
Note:

- The ***empty* positions** are **Hamming check bits** that will be **computed** next !!!

3. ***Compute*** the **values** of the **bits** at the **inserted Hamming code positions** by:

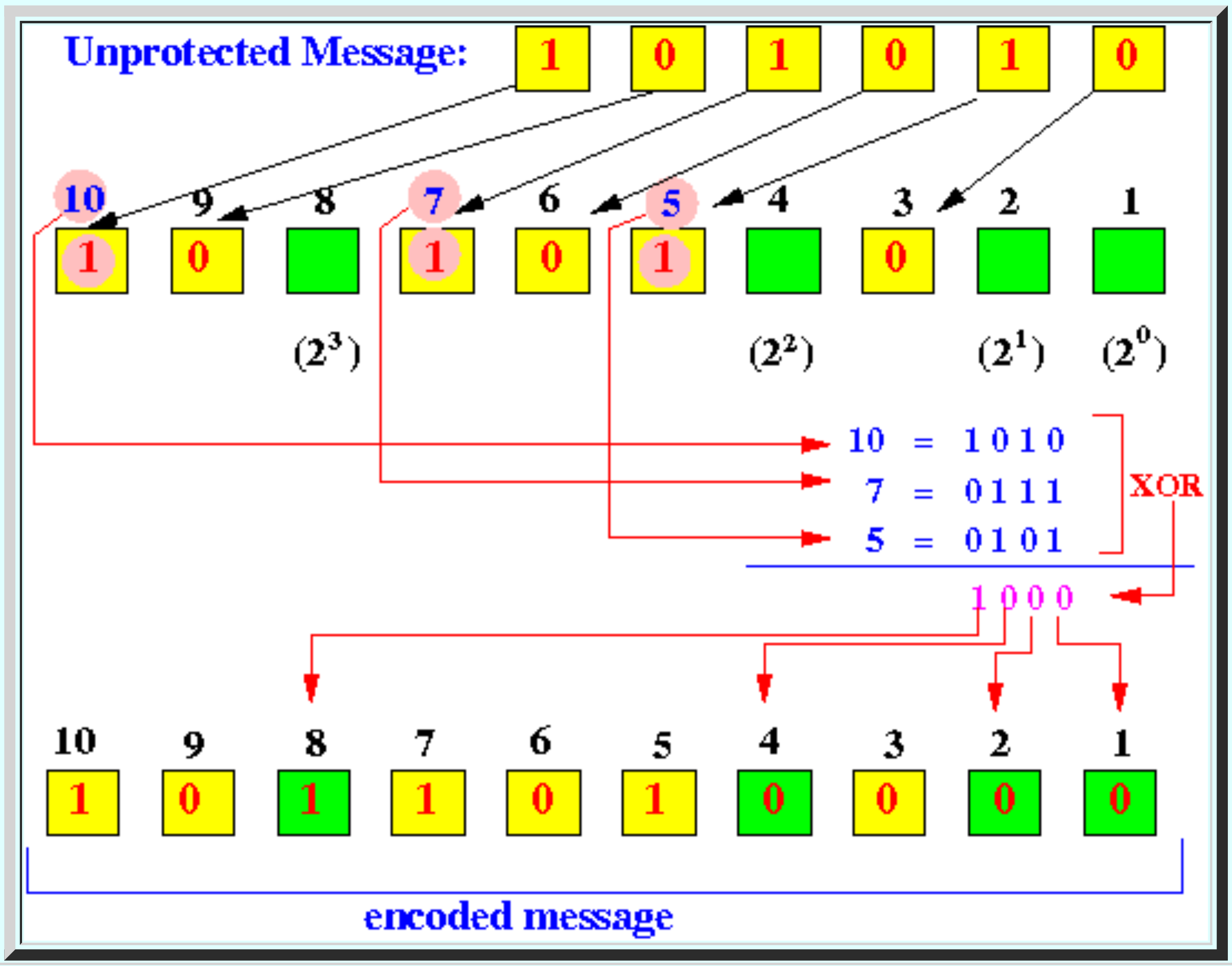
   - Converting the **index** of the **"1" bits** of the **input data** into a **binary number**
   - Perform the **XOR operations** ***column-wise*** on **all the (binary) numbers**

Example:



4. **Distribute** the **computed (check) values** into the **inserted (Hamming) bit positions**

   Example:



- **Summary (of the above):**

  - If the **sender** wants to **transmit** **101010** using **Hamming Code**, it will **transmit**:
    - **10110100000**

Hamming code: decoding procedure

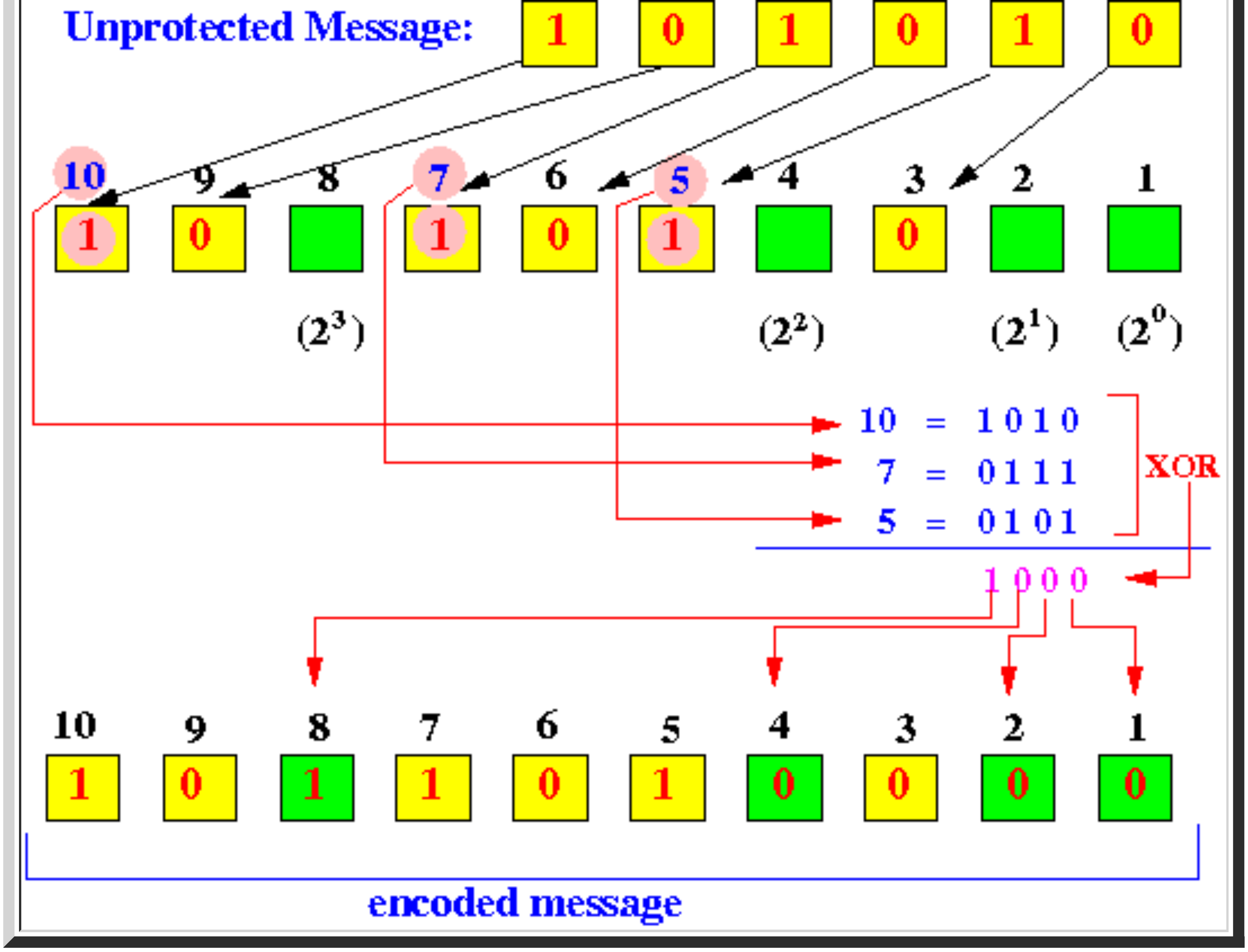
- **Decoding a Hamming coded message**
  - **Checking the *correctness* of a *received message*:**

    - Converting the **index** of the **1 bit** locations into a **binary number**
    - Perform the **XOR operations** on **all resulting binary numbers**

    - **If** the **result = 0** then the **messages (most likely)** contains **no error**
      - The **decoded message** consists of the **received bits minus** the bits at at **positions**: 2^0, 2^1, 2^2, ..., 2^k
    - **If** the **result <= length of the message** then the **messages (most likely)** contains **1 error**
      - **Change** the **value** of the **bit** at the **position** given by **"result"** (computed above)
      - The **decoded message** consists of the **received bits minus** the bits at at **positions**: 2^0, 2^1, 2^2, ..., 2^k
    - **If** the **result > length of the message** then the **messages (most likely)** contains **multiple errors**
      - The **receiver cannot perform correction**
      - It will **reject** the **message** as **corrupted**

  - **Example: 101010**

    From the discussion above:

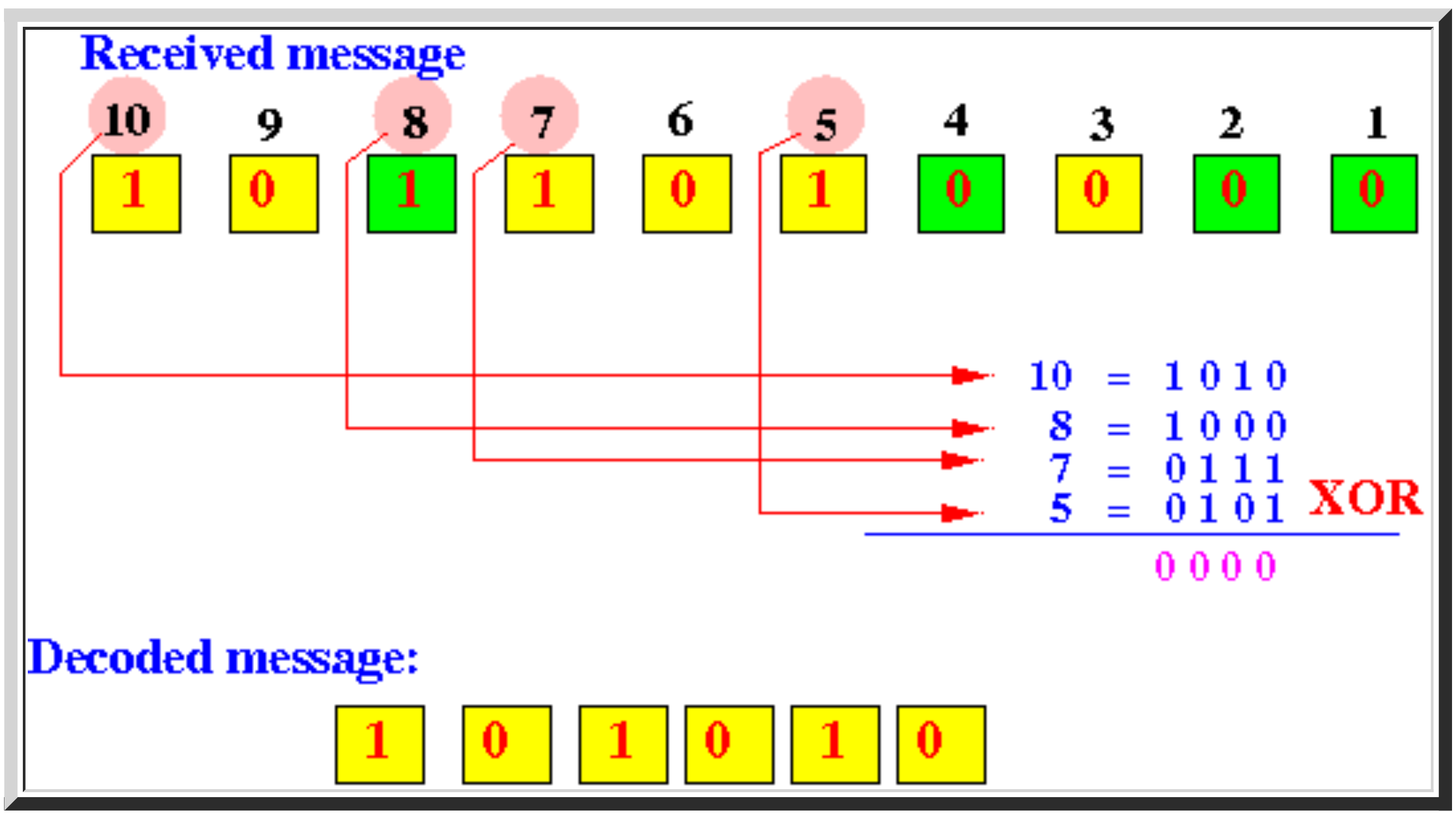


we know that the **sender** will **transmit**: **1011010000**

- **Case 1:** transmission was received with **no error**

  - **Received message = 1011010000**

  Outcome of the **decode procedure**:



Result:

- **XOR** result = **0**

  Conclusion:

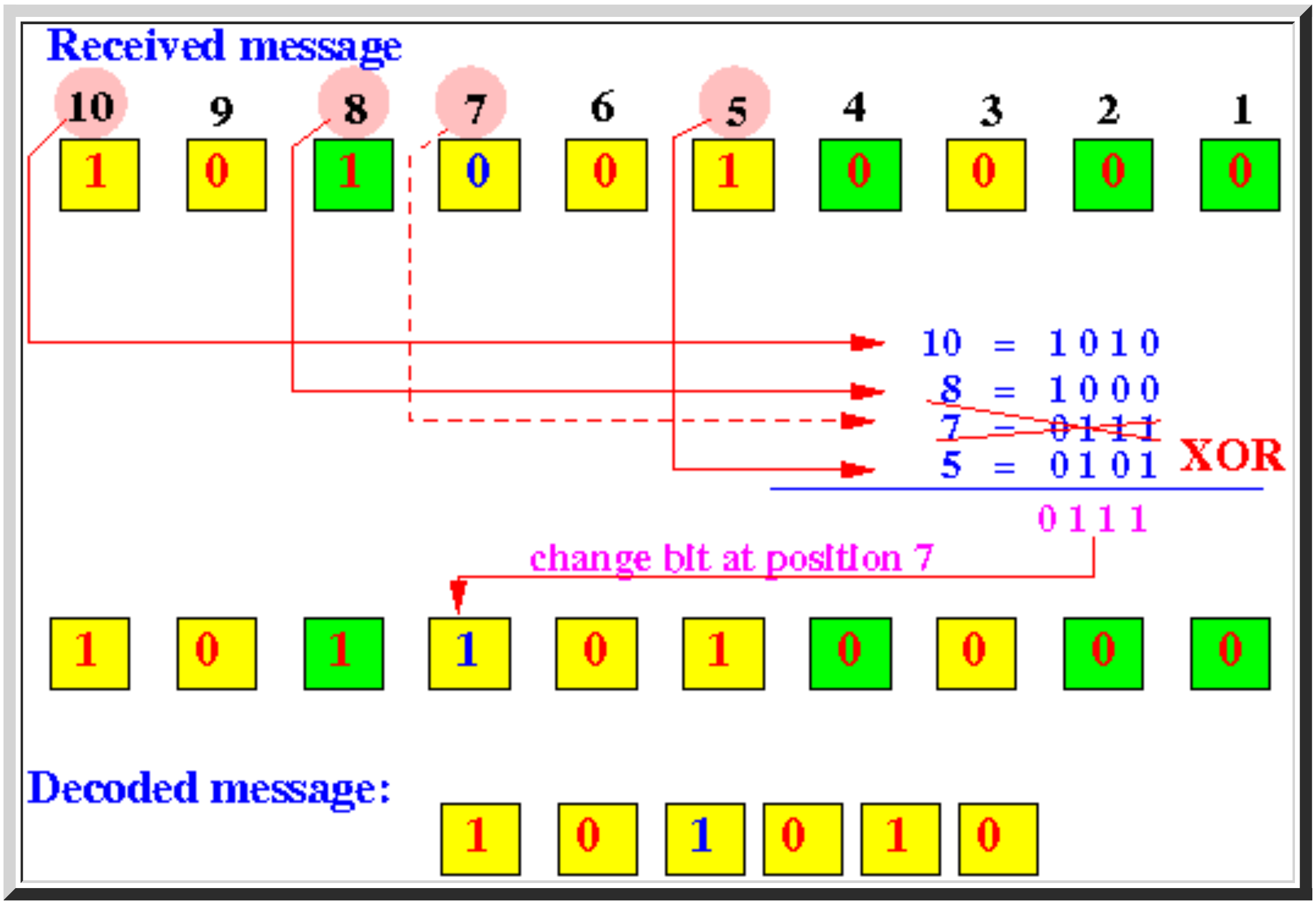
  - **Message** has **no error** (which was what happened !)

- **Case 2:** transmission was received with **one bit error**

  - **Received message = 1010010000**

    (bit position **7** was received in **error**)

  Outcome of the **decode procedure**:



  Result:

  - **XOR** result = **0111** = **7** (**≤ 10** (message length))

    Conclusion:

    - **Message** has **1 bit error** at **bit position 7** (which was what happened !)

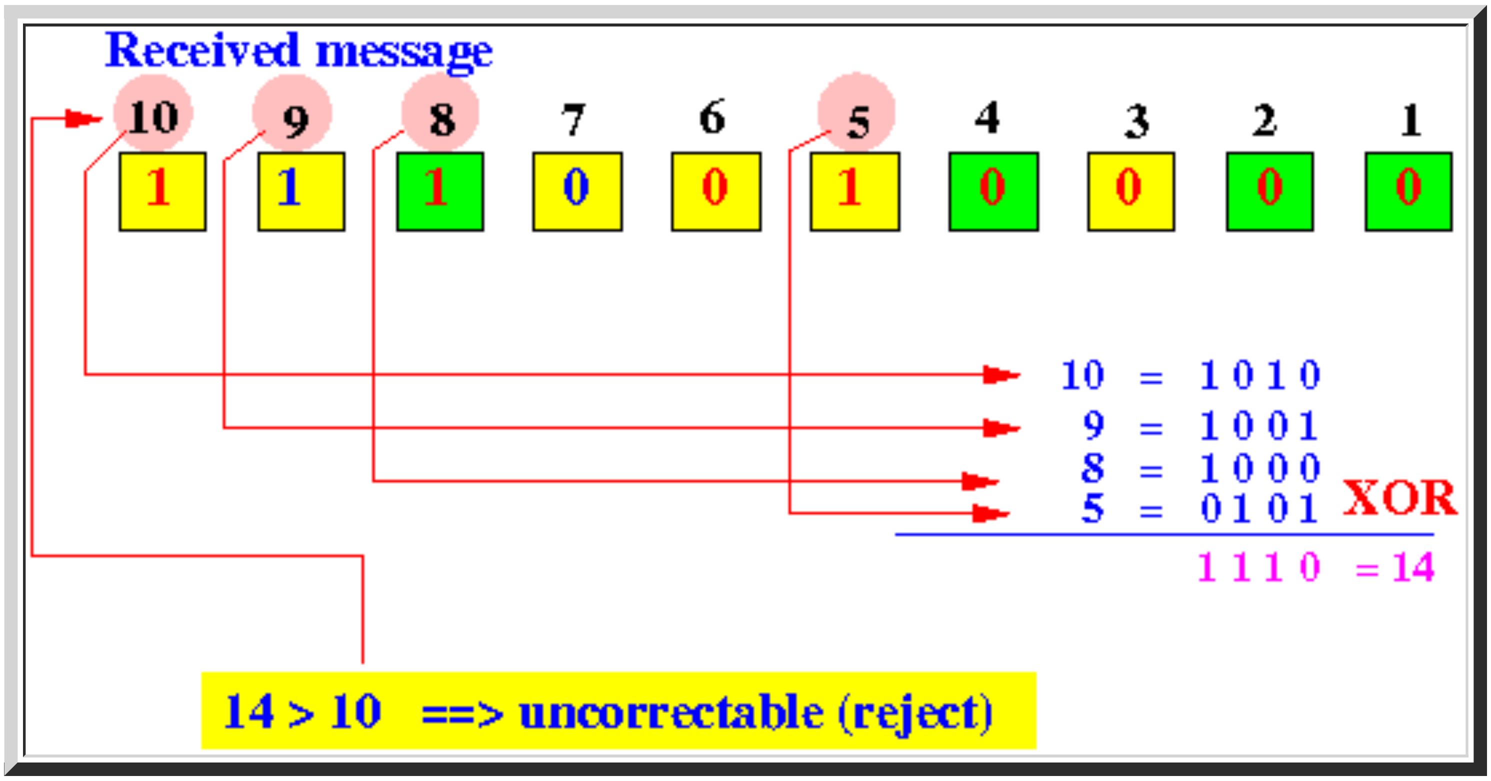
  (Try with a bit error in some ***other*** position !)

Case 3: transmission was received with (**detectable**) **two bits error**

- **Received message** = 1110010000

  (Bit positions **7** and **9** were received in **error**)

Outcome of the **decode procedure**:



Result:

- **XOR** result = **1110** = **14** (**> 10** (message length))

  Conclusion:

  - **Message** has ***multiple*** **bit errors** -- correcting is ***impossible*** (which was what happened !)

- **Warning: *multiple* error can cause *errorneous* correction in Hamming procedure**
  - **Fact:**
    - ***Some*** **multiple bit errors** will cause the **Hamming decode procedure** to believe that there was a **single bit error**

  - **Example:**
    - We use the ***same*** example:

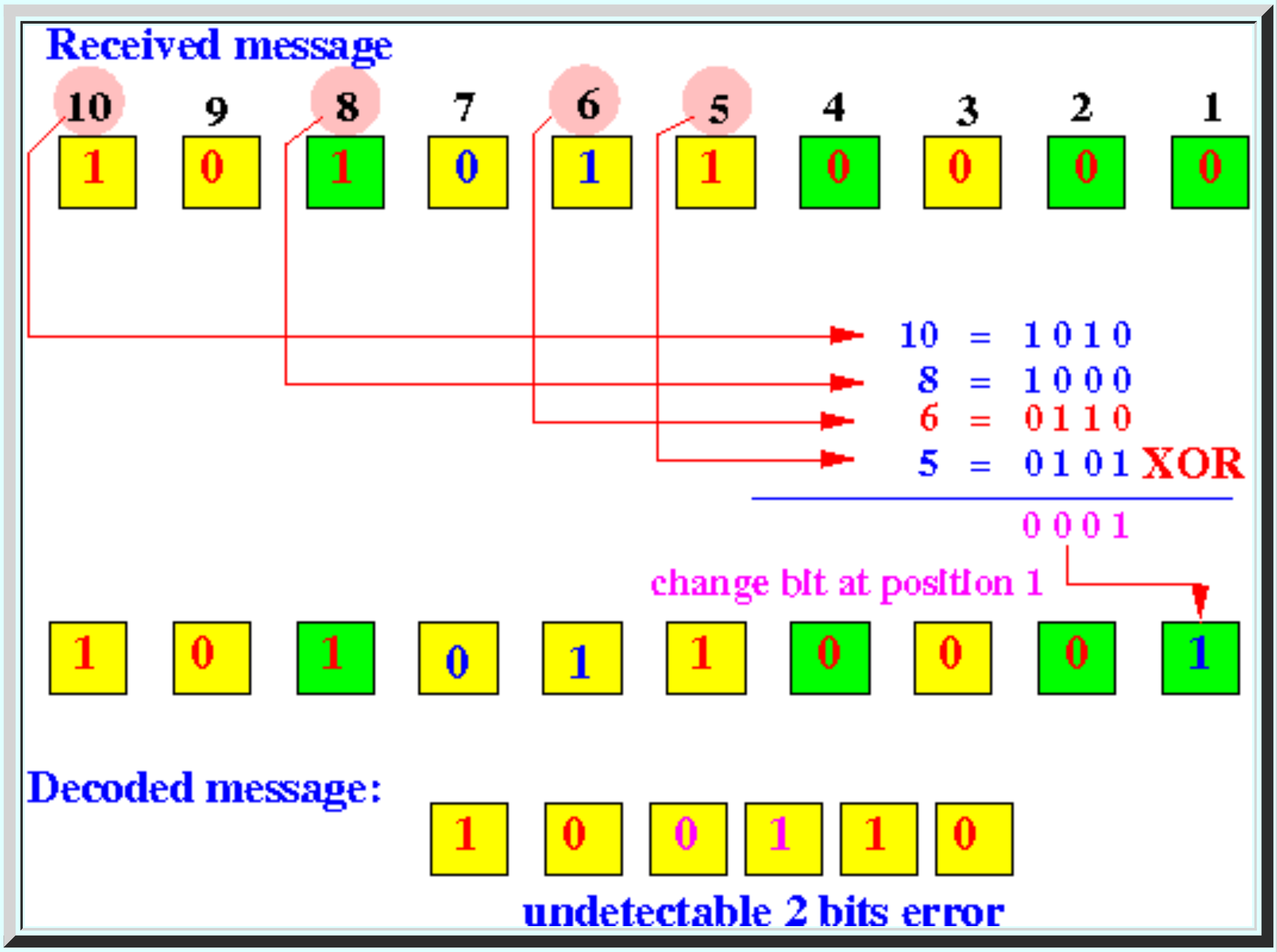
      ```
      Hamming encode message:   1011010000

      (original message was: 101010
      ```

- Suppose bit positions **6** and **7** were received in **error**:

```
Received message = 1010110000
```

- **Outcome** of the **decode procedure**:



Result:

- **XOR** result = **0001** = **1** (**≤ 10** (message length))

  Conclusion:

  - **Messages** has **1 bit error** at bit position **1**...

    (This is **wrong** !!!)

Intro to *check summing*

- **Check summing**
  - **Interpret** a **binary pattern** as **numbers**:
    - **Facts** about **binary numbers**:
      - **Computer communication always** transmit **binary codes**

      - A **binary code** can **represent** some **fact**

        Example:

```
ASCII code:

      01000001         'A'

----------------------------------------

Marital status code:

      00000000         Single
      00000001         Married
      00000010         Divorced
      00000011         Widowed
```

    - **Binary codes** can **always** be **interpreted** as an **integer**:

```
 Binary code       meaning in ASCII code       meaning as bin number
-------------    --------------------------  ----------------------
 01000001            the letter 'A'             decimal value 65
```

  - **Check summing** a **(input) message**:
    - We **interpret** a **binary code/pattern** in the **input data** as:
      - ***Integer* value** (in **binary number representation**)

    - We can **compute** the **sum** of ***all*** **value** in the **input data**:
      - We **truncate** the **(running) sum** to ***n*** **bits** to **prevent "overflow"**...
    - **Commonly used lengths** for the **summation**:
      - **8 bits**,
      - **16 bits** or
      - **32 bits**

- We *append* the (check) sum at the end of the input data

  (So the receiver can verify the (check) sum of the *original* input)

- **Example:**

```
Message:          ABC

In Binary code:   01000001   01000010   01000011

Compute check sum:

                  01000001
                  01000010
                + 01000011
               ------------
                 110000110      (truncated to 8 bits)

Transmitted message:

                  01000001   01000010   01000011   10000110
```

- **Detecting bit errors with check sums**

  - Checking for errors with check sum:

    - The receiver *knows* that the message contains a check sum (byte) at the *end* !!!

    - **Check sum verification:**

      1. The receiver compute the check sum of the received message (excluding the *last* byte)

      2. Thereceiver compares the *computed* check sum against the *received* check sum

  - **Example 1:** *no* bit errors

```
Original message:      01000001   01000010   01000011

Transmitted message:   01000001   01000010   01000011   10000110

Received message:      01000001   01000010   01000011   10000110
                                                            ^
Check sum:                                                  |
                    01000001                                |
                    01000010                                |
```

```
             01000011 +                                    |
          --------------                                   |
            110000110                                      |
                ^                                          |
                |                                          |
                +------------------------------------------+
                            Compare/verify

Check sum is equal ==> assume no bit errors
```

- **Example 2:** ***some* bit errors**

```
Original message:     01000001  01000010  01000011
Transmitted message:  01000001  01000010  01000011  10000110
--------------------------------------------------------------

Received message:     01010001  01000110  01000011  10000110
                                                       ^
Check:          01010001                               |
                01000110                               |
                01000011 +                             |
             --------------                            |
               110011010                               |
                   ^                                   |
                   |                                   |
                   +-----------------------------------+
                            Compare: 10011010 ≠ 10000110

Check sum is NOT equal ==> detects bit errors
```

- **Example 3:** the **check sum** method can ***miss* detection** of some **2 bit errors**

```
Original message:     01000001  01000010  01000011
Transmitted message:  01000001  01000010  01000011  10000110
--------------------------------------------------------------

Received message:     01000011  01000010  01000001  10000110
                                                       ^
Check:          01000011                               |
                01000010                               |
                01000001 +                             |
             --------------                            |
               110000110                               |
                   ^                                   |
                   |                                   |
                   +-----------------------------------+
                            Compare

Check sum is equal ==> assume no bit errors
```

- **An *alternate* sum operation: XOR (instead of ADD)**

  - **Goal** of the **check sum**:

    - The **check sum** is used to **protect** the **integrity** of every **(data) value** in the **entrie message**

- The *sum* computed is ***meaningless*** (i.e., the **sum** is **not** a **total** of some **useful value**)

- A **more *efficient* operation** used to **protect** the **integrity** of **serie of values** is:

  - **XOR**

  Example:

```
Message:     ABC

In Binary:  01000001  01000010  01000011

Check "sum":
                 01000001      (XOR is computed per column)
                 01000010
             XOR 01000011
            --------------
                 01000000

Transmitted message:

              01000001  01000010  01000011  01000000
```

  Note:

  - **XOR** applied to ***multiple*** **input value** works like this:
    - If there are an **even number** of **1's**, then **XOR output** is **0**
    - If there are an **odd number** of **1's**, then **XOR output** is **1**

- **Property of a message that uses an XOR sum**

  - **Fact:**

    - Due to the **property** of the **XOR operation**:
      - The **number of 1 bits** in ***every*** **column** of the **XOR check sum computation** is ***even***

  Example:

```
Message:     ABC

In Binary:  01000001  01000010  01000011

Check "sum":
                 01000001      (XOR is computed per column)
                 01000010
             XOR 01000011
            --------------
                 01000000
```

Reason:

- ***If*** a **column** has an ***odd*** **number** of **1 bits**:
  - **XOR** will **result** in **1**; making the **total # 1 bits** in the **column** ***even***

- ***If*** a **column** has an ***even*** **number** of **1 bits**:
  - **XOR** will **result** in **0**; and the **total # 1 bits** in the **column** will **remain** ***even***

- **Detecting bit errors with "XOR check sums"**
  - **Checking** for **errors** with **check sum**:
    - **How** to **verify** a **received message**:
      - The **receiver** must **compute** the **XOR check sum** of the **message** ***and*** the **check sum**

    - If the **result** = **000...00 (all bits are 0)**, then:
      - The **receiver** will **assume** that the **message** contains ***no*** **bit errors**

      Otherwise:

      - The **receiver** will **assume** that the **message** contains ***some*** **bit errors**

- **Example 1:** ***no*** **bit errors** case

```
Original message:     01000001   01000010   01000011
Transmitted message:  01000001   01000010   01000011   01000000

Received message:     01000001   01000010   01000011   01000000

Check:          01000001
                01000010
                01000011
                01000000   XOR
            -------------------
                00000000

"Xor sum" is equal to 00000000 ==> assume no bit errors
```

- **Example 2:** ***some*** **bit errors** case

```
Original message:      01000001  01000010  01000011
Transmitted message:   01000001  01000010  01000011  01000000

Received message:      01010001  01000110  01000011  01000000

Check:          01010001
                01000110
                01000011
                01000000   XOR
            -------------------
                00010100

Check sum is NOT equal to 00000000 ==> assumes some bit errors
```

- **Example 3:** the **XOR sum** method can ***also*** ***miss* detection** of some **2 bit errors**

```
Original message:      01000001  01000010  01000011
Transmitted message:   01000001  01000010  01000011  01000000

Received message:      01000011  01000010  01000001  01000000

Check:          01000011
                01000010
                01000001
                01000000   XOR
            -------------------
                00000000

XOR sum is equal to 00000000 ==> assumes no bit errors
```

Pylonimial arithmetic in modulo 2

- **CRC: a more *robust* (error detection) check summing technique**
  - **Cyclic Redundancy Check (CRC):**
    - **Cyclic Redundancy Check** = a **check "sum"** computation technique based on ***division*** of:
      - The **message (bits)** **and**
      - A **"CRC polynomial"**

    - The **Cyclic Redundance Check (CRC)** method is also known as
      - **polynomial code checksum**

    - Wikipedia page: **click here**

  - **Facts** on **CRC**:
    - The **Cyclic Redundancy Check (CRC)** method is the **most commonly used** **"*check summing*" method** in use today

    ---

    - When a **communication professional** use the term "**check sum**" he/she is **referring** to a **CRC check sum**

  Why is **CRC** is **so *popular***:
    - **CRC codes** can **detect common occuring errors** that are **caused** by ***noise* (lightning)** in **transmission channels**.
      - **CRC** can **detect *n* consecutive bit errors**

    - **CRC *encoders/decoders*** are ***easy* to construct**:
      - **Electronic circuits** used to **compute** the **CRC code** consists of **handful** of **elementary *gates*** and **1-bit memery circuits (D-flipflops)** !!!

- **CRC computation**
  - **Fact:**
    - The **CRC check sum computation** uses:
      - **Polynomial arithmetic** in **modulo 2** arithmetic
  - **Furthermore:**
    - The **polynomials** will be ***represented*** using ***bit* strings**.
    - **Polynomial arithmetic operation** becomes ***simple*** **binary number operations using XOR** !!!
- **Preliminary to CRC computation**
  - We will **discuss** the follwoing **material first**:
    1. **Polynomials** representation in **modulo 2 arithmetic**
    2. **Performing modulo 2 arithmetic** on **polynomials**
    3. **Representing polymonials** using **bit strings**
    4. **Performing modulo 2 arithmetic** on **polynomials** represented in **bit strings**
- **Polynomial representation using modulo 2 arithmetic**
  - **Modulo 2 representation:**
    - An ***even* coefficient** is **mapped** to **0**
    - An ***odd* coefficient** is **mapped** to **1**
  - **Example** representing **polynomials** in **modulo 2**:

```
Polynomial:                                -x^3 + 2x + 1
The polynomial in module 2 representation:  x^3      + 1   (mod 2)

  (Coefficient -1 is  odd and is mapped to 1)
  (Coefficient  2 is even and is mapped to 0)
  (Coefficient  1 is  odd and is mapped to 1)
```

- Polynomial arithmetic using modulo 2 arithmetic
  - Example: addition

```
   (x^3 + x)  +  (x + 1)  =  x^3 + 2x + 1
                          =  x^3 + 1          (mod 2)
```

  - Example: subtraction

```
   (x^3 + x)  -  (x + 1)  =  x^3  -  1
                          =  x^3  +  1          (mod 2)
```

  - Example: multiplication

```
   (x^2 + x) × (x + 1)  =  (x^3 + x^2) + (x^2 + x)
                        =  x^3 + 2x^2 + x
                        =  x^3 + x            (mod 2)
```

  - Example: division

```
                   x^2 + 1
                ----------------
       x + 1 /     x^3 + x^2 + x
                   x^3 + x^2
                 ---------
                            x
                            x + 1
                          ------
                               -1

   x^3 + x^2 + x
  ------------ = x^2 + 1   remainder  1      (mod 2)
     x + 1
```

- Modulo 2 arithmetic

- Summary: **modulo 2** ***arithmetic***

```
   0        1         0        1
 + 0      + 0       + 1      + 1
-----    -----     -----    -----
   0        1         1        0
```

Observation:

- The **+ (mod 2)** operation is the ***same*** as an ***XOR*** **(exclusive OR)** operation used in **logic** !!!!

- **Polynomial representation using a bit string**

  - **Bit positions**

    - **Bit string** and **position** of **a bit** in the **bit string**:

```
Bit string:      1010
                 ^^^^
                 ||||
                 |||+-- bit position 0
                 ||+--- bit position 1
                 |+---- bit position 2
                 +----- bit position 3
```

  - **Bit string** representation of a **polynomial**:

    - The **bit** at **position** ***k*** is the ***coefficient*** of the **factor** x^k in the **polynomial**

Example:

```
Polynomial:
                 x^5 + x^4 + x^2 + 1

                  x^5 +  x^4 +         x^2 +        1
            =  1x^5 + 1x^4 + 0x^3 + 1x^2 + 0x^1 + 1x^0
```

```
Bit string representation for the polynomial:

                    110101
                    ^^^^^^
                    ||||||
                    |||||+----- x^0
                    ||||+------ x^1
```

```
|||+-------- x^2
||+--------- x^3
|+---------- x^4
+----------- x^5
```

- **Polynomial arithmetic using modulo 2 arithmetic with bit strings**
  - **Polynomial arithmetic** in **modulo 2** can be **perform** with **bit string representation** as **follows**:
    - **Replace** the **+ (mod 2) operation** with:

      ```
      XOR
      ```

  - **Example: addition**

```
(x^3 + x)   +   (x + 1)   =   x^3 + 2x + 1
                          =   x^3 + 1           (mod 2)

In bit string representation:

   1010 + 0011 ==>   1010
                     0011
                    ------ XOR
                     1001  ===> x^3 + 1    (mod 2)

Same result !!!
```

  - **Example: subtraction**

```
(x^3 + x)   -   (x + 1)   =   x^3  -  1
                          =   x^3 + 1           (mod 2)

In bit string representation:

   1010 + 0011 ==>   1010
                     0011
                    ------ XOR
                     1001  ===> x^3 + 1    (mod 2)

Same result !!!
```

- **Example: multiplication**

```
   (x^2 + x) × (x + 1)  =  (x^3 + x^2) + (x^2 + x)
                        =  x^3 + 2x^2 + x
                        =  x^3 + x            (mod 2)

   In bit string representation:

       110 × 011   ==>    110
                        × 011
                        -----
                          110
                         1100
                        00000
                      --------- XOR
                         1010  ====>  x^3 + x   (mod 2)

   Same result !!!
```

- **Example: division**

```
                 x^2 + 1
              ----------------
       x + 1 /   x^3 + x^2 + x
                 x^3 + x^2
               ---------
                          x
                          x + 1
                        ------
                             -1

     x^3 + x^2 + x
    ------------- = x^2 + 1   remainder  1       (mod 2)
       x + 1

   In bit string representation:

                 101        ====> Quotient = x^2 + 1   (mod 2)
              ---------
          11 /  1110
                11||
               ---v|
                 01|
                 00|
               ----v
                  10
                  11
               -----
                   1    ====> remainder = 1 (mod 2)
```

- **Conclusion:**

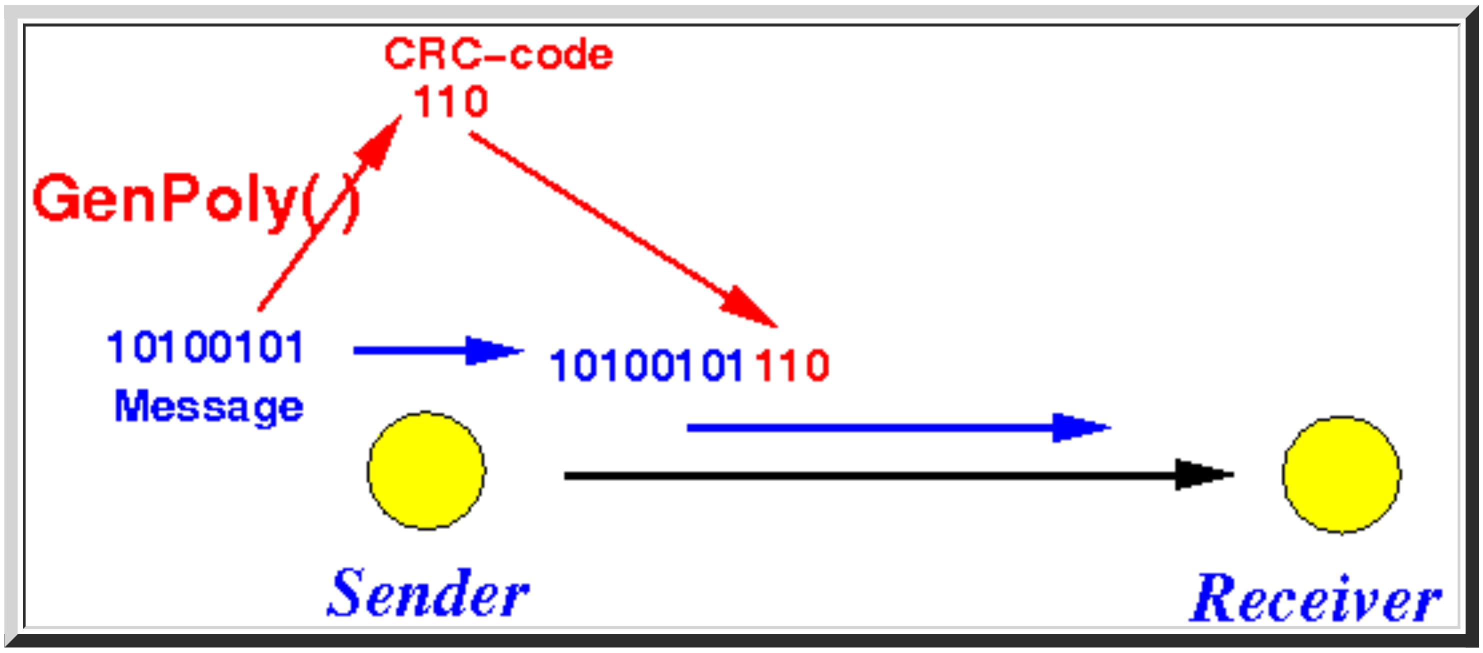
  - We can **use** the ***bit string** representation* an the **XOR operation** to **perform** the **compution** for the **polynomial** ***arithmetic***

CRC encoding procedure

- **Generator polynomial**
  - **Generator polynomial:**
    - **Generator polynomial** = the ***divisor*** **polynomial** in the **polynomial division** operation

  - **Obvious fact:**
    - The **sender** and the **receiver** must use the ***same*** **generator polynomial** to **encode**/**decode** the **messages**.

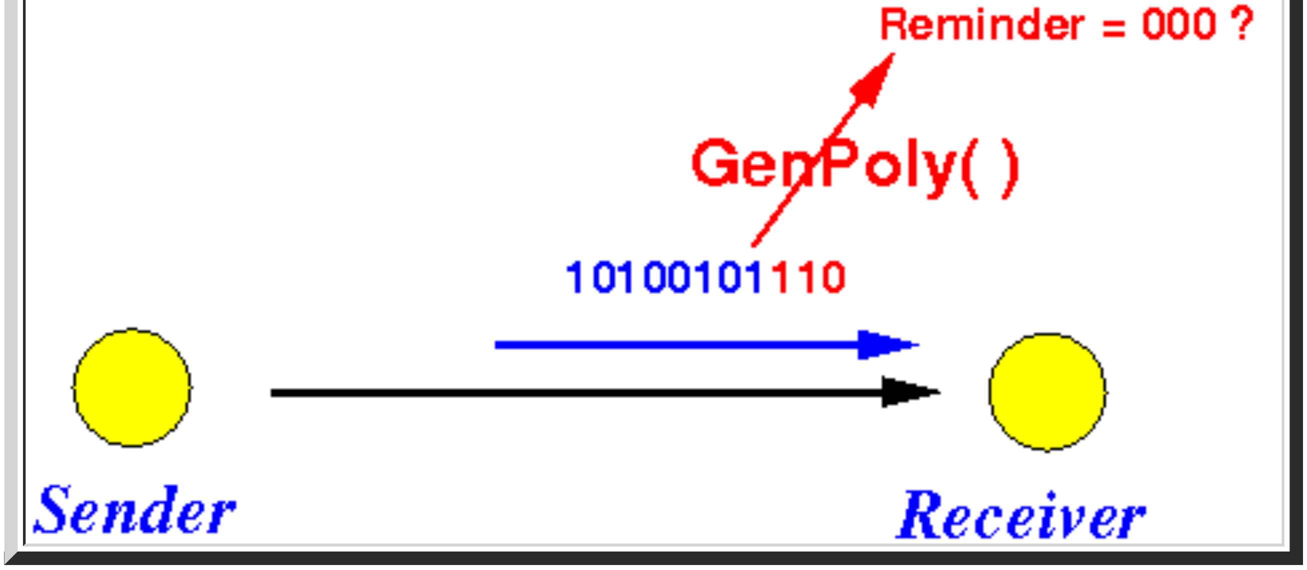
- **How to use CRC code**
  - **How** the **sender** transmits a **message**:



Explanation:

- The **sender** ***divides*** the **message** by the **generator polynomial**
- The **sender** ***includes*** the ***remainder*** of the **division** in the **transmitted message**

  - **How** the **receiver** **checks** a **received message** for **correctness**:



Explanation:

- The **receiver divides** the **message + CRC code (remainder)** by the ***same*** **generator polynomial**

- The **receiver** will ***assume*** there is **no bit error(s)** if the **remainder** of the **division** is **equal** to **0**

- ***Encoding procedure*** **of the Cyclic Redundancy Check (CRC) codes**
  - The **CRC encoding procedure**:

1. ***Let***:

```
N = length of the generator polynomial
  = number of bits in the generator polynomial
```

---

2. **Append** ***N*–1** **ZERO (0) bits** to the ***end*** of the **input message**.

---

3. **Divide** the **messages + (N-1) ZEROs** by the **generator polynomial**.

   Note:

   - The **division operation** will use the **XOR operation** as the **subtract operation**

---

4. **Append** the **remainder** of the **"division"** to the **input message**

   - The ***resulting*** **message** is the **CRC** ***protected*** **message**

- **Example**: computing the **CRC code** (= **check sum**) for a **message**

```
Generator polynomial: x^3 + x^2 + 1

Genenrator polnomial in bit string: 1101

   (1101 represents the polynomial: 1×x^3 + 1×x^2 + 0×x^1 + 1×x^0)

-----------------------------------------------------------------

 N = 4 (because Generator polynomial has 4 terms, or bits in "1001")

-----------------------------------------------------------------

Input message:                x^4 + 1

Input message in bit string: 10001

-----------------------------------------------------------------

Computing the CRC (check sum) for the message 10001 using Gen Polyn 1101:

   1. Add N-1 (= 3) ZERO's bits to message:

         Message + 3 ZEROs = 10001000

   2. Compute CRC by dividing "message + 3 zeros" by generator polynomial:

                     00011
                   ----------------
               1101 /   10001000
                        1101  <----- **** see Note below !! ****
                        ---- (XOR)
                         1011
                         1101
                         ---- (XOR)
                          1100
                          1101
                          ---- (XOR)
                          000100 <---- remainder = 100

                CRC = 100

  3. CRC protected message = 10001100
```

- **Note:** ***how to*** **perform** the **"division"**

  - **Whenever** the ***leading*** **bit** of the **divident** **=** **1**, you get a **1** in the **quotient**.

Example:

```
              1
         ------------
   1101 / 10001000        // Even when 1000 < 1101 !!!
          1101
         ----- XOR
          0101
```

- The **division** uses the **XOR operation** as ***subtraction***

- **A longer example....**
  - **Example:**

```
   CRC generator polynomial:  11011          (N = 5)
   Message:                   11100110

   1. Add  4  0-bits to the message:

             11100110  -->    111001100000

   2. Divide:

                   10101110
              ---------------
       11011 / 111001100000
               11011||||||
              ------v|||||
                 1111|||||
                 0000|||||
               ------v||||
                 11111||||
                 11011||||
                ------v|||
                   1000|||
                   0000|||
                  -----v||
                   10000||
                   11011||
                  ------v|
                    10110|
                    11011|
                   ------v
                     11010
                     11011
                    ------v
                      00010
                      00000
```

```
                       -----
                        0010

   3. CRC protected message = 111001100010
```

- **Property of a CRC protected message**
  - **Recall**: **how** is the **CRC code computed**

```
                    ------------------------------
    CRC-polynomial /  Original-Message + 000..000
                      ....
                      -----------------------------
                                          CRC-code
```

  - A **CRC protected message** consists of:

```
    Original-Message + CRC-code
```

  - **Property** of **every** **CRC protect message**:
    - The **remainder** of a **CRC protected message** divided by the **CRC polynomial** (used in the **encoding**) is:
      - **Always** equal to **ZERO** !!!

      In formula:

```
   (Original-Message + CRC-code) % CRC-polynomial == 0 !!!
```

Reason:

```
Since:

                    ------------------------------
    CRC-polynomial /  Original-Message + 000..000
                      ....
                      -----------------------------
                                          CRC-code

We will have:

                    ------------------------------
```

```
CRC-polynomial /  Original-Message + CRC-code
                  ....               ||||||||
                                     VVVVVVVV
                  -----------------------------
                                     CRC-code
                                 XOR CRC-code
                              -----------------
                                     000..000
```

(The **dvision** will **now** bring down the **value "CRC-code"** which will be **subtracted** from the **remaining value "CRC-code"** --- therefore **producting** the **final remainder 0**)

- **Example:**

  - **Previously**, we have **computed** the **CRC code** for a **message**

    ```
    CRC generator polynomial:   11011           (N = 5)

    Message:                    11100110

    CRC code =                  0010

    CRC protected message     = 111001100010
    ```

  - **CRC check:**

    ```
    Divide:

             Received message    111001100010
       by    CRC polynomial      11011

    ------------------------------------------

            (don't care about the quotient)
            ----------------
     11011 / 111001100010
             11011
            ------
               11111
               11011
              ----------
                 10000
                 11011
               ----------
                  10110
                  11011
                 -------
                   11011
                   11011
                  ------
                   000000  <--- Remainder = 0 !!
    ```

CRC decoding procedure

- **Decoding a CRC protected message**
  - **CRC *decoding*** procedure:
    - **"Divide"** the ***entire* received message** by the ***generator polynomial***

      Note:

      - The **receiver does *not*** append any **ZERO (0) bits** to the **received message** !!!

    - There are **2 *possible* outcomes** of the **"division"**:
      - The ***remainder*** of the **"division"** is **equal** to **0000..000**:
        - The **message** will **assumed** to be ***free* of errors**
      - **Otherwise** (**remainder** is ***not* equal** to **000..00**):
        - The **message** is **assumed** to **contain** some **bit errors**

  - **Example 1:** received message has **no errors**

```
Generator polynomial: 1101  (Math. notation: 1×x^3 + 1×x^2 + 0×x^1 + + 1×x^0)

N = 4 (Generator polynomial has 4 terms, or bits in "1001")

Sender: (from a previous example)

              Message:                  10001
              CRC protected message:   10001100

1. Received message = 10001100        (error free !!!)

2. Check CRC by dividing "received message" by generator polynomial:

                   00011
                 ----------------
```

```
                 1101 /   10001100
                          1101
                          ---- (XOR)
                           1011
                           1101
                           ---- (XOR)
                            1101
                            1101
                            ---- (XOR)
                               000 <---- remainder = 000

3. Received message is assumed to be correct

   Actual message = 10001   (with the CRC bits removed)
```

- **Example 2:** received message has **two bits in error**

```
Generator polynomial: 1101  (Math. notation: 1×x^3 + 1×x^2 + 0×x^1 + + 1×x^0)

N = 4 (Generator polynomial has 4 terms, or bits in "1001")

--------------------------------------------------------------

Sender: (from a previous example)

              Message:                10001
              CRC protected message:  10001100

--------------------------------------------------------------

1. Received message = 11101100

2. Check CRC by dividing "received message" by generator polynomial:

                          00011
                        ----------------
                 1101 /   11101100
                          1101
                          ---- (XOR)
                           01111
                            1101
                            ---- (XOR)
                              1000
                              1101
                             ---- (XOR)
                              0101  <---- remainder = 101 ≠ 000 !!!

3. Received message is assumed to be corrupted

   Bit errors have been detected !!!
```

The effect of errors on a CRC message

- **Preliminaries**

  - **Recall** from the **CRC decoding** procedure:

    - Let **M** be **CRC protected message**
    - Let **G** be the **generator polynomial**

    - If the **message M** is **received** with ***no* error**, then:

    ```
         --------
      G /  M
          .....

         Remainder = 0000...000
    ```

- **The error polynomial**

  - **Error polynomial**:

    - **Error polynomial** = the **polynomial** that **represents** the **bit errors** in the ***received* message**

      - **If** a **bit** is **received *correctly***:

        - The **corresponding bit** in the **error polynomial** = **0**

      - **If** a **bit** is **received *incorrectly***:

        - The **corresponding bit** in the **error polynomial** = **1**

  - **Example:**

    ```
    Transmitted message = 10001100

    Received message    = 11101100

                          ||||||||
                          VVVVVVVV

    Error polynomial    = 01100000
    ```

  - **How to determine** the **error polynomial**:

```
Error = ReceivedMsg XOR TransmittedMsg
```

- **The CRC computation and the error polynomial**
  - **Claim:**
    - The **CRC remainder** computed using:

      ```
      Received Message / generator polynomial
      ```

      is **equal to** the **CRC remainder** computed using:

      ```
      Error / generator polynomial
      ```

  - **Proof:**

```
  Error = ReceivedMsg XOR TransmittedMsg

Therefore:

  Error % GP = (ReceivedMsg XOR TransmittedMsg) % GP    (GP = Gen Poly)

             = (ReceivedMsg % GP) XOR (TransmittedMsg % GP) // TransmittedMsg % GP = 0 !!!

             = (ReceivedMsg % GP) XOR 0

             = ReceivedMsg % GP
```

  - **Example:** (previous example)
    - **Message** with **bit errors**:

```
Generator polynomial: 1101  (Math. notation: 1×x^3 + 1×x^2 + 0×x^1 + + 1×x^0)

N = 4 (Generator polynomial has 4 terms, or bits in "1001")
```

```
Sender: (from a previous example)

              Message:                  10001
              CRC protected message:    10001100
```

```
1. Received message = 11101100

2. Check CRC by dividing "received message" by generator polynomial:

                          00011
                        ----------------
                  1101 /  11101100
                          1101
                          ---- (XOR)
                           01111
                            1101
                            ---- (XOR)
                              1000
                              1101
```

```
                                ---- (XOR)
                                 0101  <---- remainder = 101 ≠ 000 !!!

3. Received message is assumed to be corrupted

   Bit errors have been detected !!!
```

- We can **obtain** the ***same*** **CRC remainder** as **follows**:

```
1. Received message = 11101100
                    = 10001100 + 01100000
                                 ^
                                 |
                          "error polynomial"

2. Check CRC by dividing the error polynomial by generator polynomial:

                              1001
                        ----------------
                  1101 /   01100000
                            1101
                            ----
                               1000
                               1101
                              -----
                                101  <---- same remainder = 101 !!!
```

- **When can the CRC method detect error ?**
  - **Recall** that:

    - The **CRC method** has **detect** (some) **bit errors** ***if***:
      - The ***remainder*** of the **check procedure ≠ 0000..0000**

    - **In other words:**

```
    if ( ReceivedMsg % GenPolynome  ≠   0000...0000 )
       then:  bit errors has been detected
```

  - From **above**:

```
   ReceivedMsg % GenPolynome  =  Error % GenPolynome  !!!
```

Therefore:

```
    if ( Error % GenPolynome  ≠  0000...0000 )
       then:  bit errors has been detected
```

In other words:

- If the **error polynomial** is ***not* divisible** by the **generator polynomial**, then:

  - The **bit error(s) will** be **detected** by the **CRC method**

    (**Because** the ***remainder*** of the **division** is **≠ 000...0000** !!!)

  Otherwise:

  - The **bit error(s) will not** be **detected** by the **CRC method**

    (**Because** the ***remainder*** of the **division** is **= 000...0000** !!!)

- **Example:**

```
Generator polynomial: 1101  (Math. notation: 1×x^3 + 1×x^2 + 0×x^1 + + 1×x^0)

N = 4 (Generator polynomial has 4 terms, or bits in "1001")

--------------------------------------------------------------------------------

Sender: (from a previous example)

              Message:                  10001
              CRC protected message:    10001100

--------------------------------------------------------------------------------

1. Received message = 11100100
                    = 10001100 XOR 01101000

2. Check CRC by dividing "received message" by generator polynomial:

                         00011
                       ----------------
                 1101 /  11100100
                         1101||||
                         ----v|||
                          01101||
                           1101||
                           ----vv
                             0000


3. Received message is assumed to be CORRECT

   Bit errors have NOT been detected !!!
```

Designing *good* CRC generating polynmials

- **Designing CRC *good* polynomials**
  - There is a **rich Mathematical theory** on **how to design** **CRC polynomials** with **certain desirable properties**
    - It's **beyond the scope** of this ***Networking* course** to go into the **Mathematical details**.
    - I will only show you **2 results**
      1. **How to** design a **generator polynomial** that can **detect** an ***even number*** of **bit errors**
      2. **How to** design a **generator polynomial** that can **detect *n* consecutive bits** in **error**

- **Preliminary: polynomial factoring and *primitive* polynomial**
  - **Factors** of a **polynomial**
    - **When** a **polynomial** can be **obtained** by **multiplying** 2 **polynomials**:

      $$\text{polynomial} = \text{polynomial}_1 \times \text{polynomial}_2$$

      we **say** that:

      - The **polynomial** can be ***factored***
      - Polynomials polynomial_1 and polynomial_2 are the ***factors*** of the polynomial **polynomial**

Example:

$$\begin{aligned}(x^2 + x + 1) \times (x + 1) &= (x^3 + x^2 + x) + (x^2 + x + 1) \\ &= x^3 + 2x^2 + 2x + 1 \\ &= x^3 + 1 \qquad \text{(mod 2)}\end{aligned}$$

Therefore:

$x^2 + x + 1$ and $x + 1$ are factors of $x^3 + 1$

```
because:

      x^3 + 1 =  (x^2 + x + 1) × (x + 1)
```

- Primitive polynomial:

  - **Primitive polynomial** = a **polynomial** that **cannot** be **factored**

    I.e., A **primitive polynomial** **cannot** be written as a **product** of **2 different polynomials**:

    ```
    polynomial1 × polynomial2
    ```

- **Generator polynomial that can detect all *odd* number of bit errors**

  - **Parity Property:**

    - A **generator polynomial** that **contains** the **factor** `x+1` **(= 11)** can **detect *all* errors** that affect an ***odd* number of bits**

      - This **property** is **called** the **Parity property**

  Example:

```
Generator polynomial:   1001    has the factor 11

How to check if gen. polyn. contains the factor "x+1" (= 11):

Divide by 11:

          111
       --------
   11 /  1001
         11
        ---
          10
          11
         ---
           11
           11
          ---
            0  <---- That means:  1001 = 11 × 111

                     in other words: 1001 contains the factor "x+1"
```

- **Property of polynomials that contains "x+1" (= 11) as a factor**

  - **Property** of **polynomial** with **(x+1)** as a **factor**:

    - **Every message** that is **CRC protected** by a **polynomial** with **(x+1)** as a **factor** has this **property**:

      - The ***number*** of **1 bits** in the **CRC protected message** has an ***even*** **number** of **1 bits**

  - **Example:**

```
Message:          10011          (has an odd # 1 bits)

CRC polynomial:   1001           (1001 has 11 as factor)

Compute CRC code:

                 -------------
         1001 /  10011000
                 1001
                 ----
                    01000
                     1001
                    -----
                      001    <---- Remainder

CRC protected message:

               10011001   (has an even number 1 bits !!!)
```

- **Why a polynomial containing "x+1" (= 11) can detect an even number of bit errors**

  - **Why** a **polynomial** containing **"x+1" (= 11)** can **detect** an ***odd*** **number** of **bit errors**:

    - An ***odd*** **number** of **bit errors** will **result** in a ***received*** **message** containing an ***odd*** **number** of **1 bits**

      Reason:

      - The **CRC protected message** has an ***even*** **number** of **1 bits** (see previous claim)

      - There are **2 possible bit errors**

```
Bit error type                 | Effect
-------------------------------+-------------------------
0 bit sent --> 1 bit received  | # 1 bits increased by 1
1 bit sent --> 0 bit received  | # 1 bits decreased by 1
```

- Therefore:

  - **If** there are **0 bit errors**: the **received message** has an ***even* number** of **1 bits**
  - **If** there is **1 bit error**: the **received message** has an ***odd* number** of **1 bits**
  - **If** there are **2 bit errors**: the **received message** has an ***even* number** of **1 bits**
  - **If** there are **3 bit errors**: the **received message** has an ***odd* number** of **1 bits**
  - **And so on** ....

- **Fact:**

  - **When** a ***received* message** containing an ***odd* number** of **1 bits** is **divided** by the **genertor polynomial**:

    - The **remainder** of the **division** will ***not*** be **equal** to **000...000** !!!

  This makes sense, because:

  - A ***correctly* received CRC protected message** will ***always*** contain an ***even* number** of **1 bits** !!

    Therefore:

    - A **received message** with an ***odd* number** of **1 bits** can ***never*** be ***assumed* correct** !!!

- **Generator polynomial that can detect *k* consecutive bit errors**

  - **Property** of a **primitive polynomial** of **degree *n***:

    - If x^n **+ .... + 1** is a **primitive polynomial** of **degree *n***, then:

      - The **CRC code** can **detect** any ***n* or less *consecutive* bit errors**

  - **Reason:**

```
Error polynomial representing k-consecutive errors:   1111..1
                                                      <----->
```

```
                                                     k bits
  Generator polynomial =   1.........1
                           <---------->
                             n+1 bits

   The division:

                                          -----------
            1.........1  /  1111..1
            <---------->     <------>
              n+1 bits        k bits

    will have a non-zero remainder if:

                                   k ≤ n
```

- **Therefore:**

  - A **generator polynomial** containing an **primitive polynomial** of **degree *n*** can detect ***k* consecutive bit errors** ---- for any ***k* ≤ *n***).

- **Commonly used (international standard) generator polynomials**

  - **Some commonly used CRC generator polynomials:**

Name	Generator polynomial
CRC-CCITT	10001000000100001
CRC-16	11000000000000101
CRC-32	100000100110000010001110110110111

  - More **CRC** polynomials at **Wikipedia page**: click here

Implementing the CRC scheme in *hardware*

- **Implementing the CRC division algorithm with *hardware***

  - An **important advantage** of **CRC** codes is:

    - The **"divide"** algorithm used in the **CRC** can be ***easily*** implemented with ***cheap* hardware**:

      - **Shift-register** (actually: **D-flipflops**) **and**
      - **XOR** circuits

- **Shift registers**

  - **Shift-register circuit:**

    - **1 bit shift register** = a **circuit** with that **copies** the **input value** to its **output** during the time that the **clock signal** changes from **0** ⇒ **1** (or from **1** ⇒ **0**)

  - You can see the **behavior** of a **serie** of **shift-register** in this demo:

```
/home/cs455001/demo/Logic-Sim/shift-register
```

- **XOR circuit**

  - **XOR circuit:**

    - **XOR circuit** = a **circuit** with **2 inputs** and **one output**
    - The **XOR** function is as follows:

```
0  XOR  0  =  0
0  XOR  1  =  1
1  XOR  0  =  1
1  XOR  1  =  0
```

  - You can see the **behavior** of the **XOR** circuit in this demo:

```
/home/cs455001/demo/Logic-Sim/xor
```

- **Hardware CRC implementation**
  - **How to** create a **circuit** that compute the **CRC code** for a given **generator polynomial:**

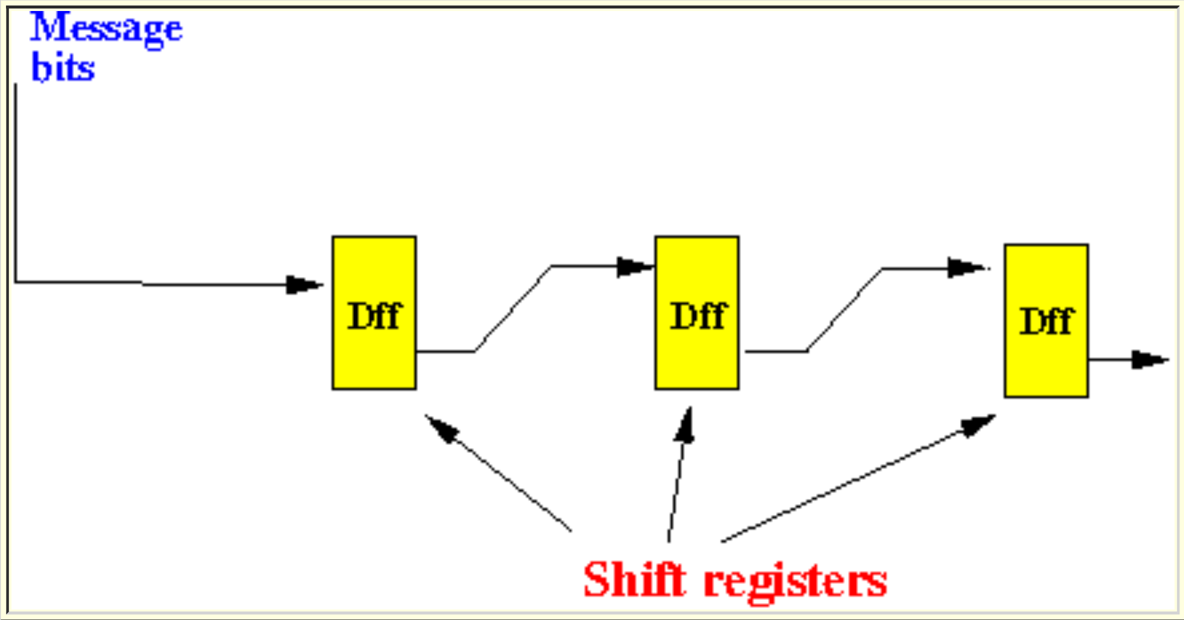
    - ***Let***

      - ***N*** = **length** of the **generator polynomial**

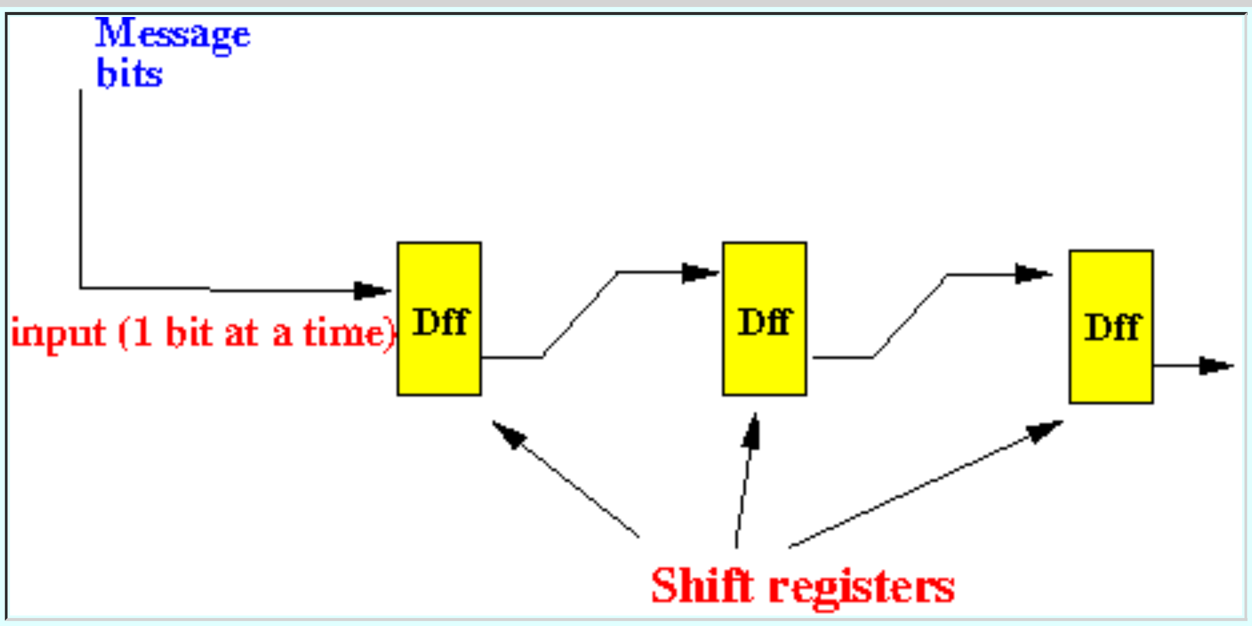
    - Use **(*N*−1) D-flipflops (= shift registers)** to **store** the **remainder of the division**.
    - **Start** by

      - Putting **(*N*−1) Dffs** in a **row**.
      - **Connect** the **output** of the ***ith*** **Dff** to the **input** of the ***i+1th*** **Dff**

        Example: (***N* = 4** or ***N*−1 = 3**)



    - The **input** of the **first Dff** will **receive** the **message *one* bit at a time**:



    - We must **modify** the **circuit** as follows:

      - Read the **CRC generator polynomial** in the ***reverse* order**.

- In **this step**, you will **only** use ***N*−1** bits of the **CRC generator polynomial**

- For **bit *i*** in the **(reversed ordered)** **CRC generator polynomial** do:

  - if **bit *i* = 1**, then:

    - **Change** the ***direct* connection** to the **input** of the **Dff** (**shift register**) to an **XOR gate**

      - **I.e.:** ***insert*** an **XOR circuit** **before** the **i^{th} Dff**

    - **Connect** the **output** of the ***last*** **Dff** as the ***second*** **input** of the new **XOR circuit**

      - **I.e.**: the **output** of the ***last*** **Dff** is **fed back** into the **i^{th} Dff**)

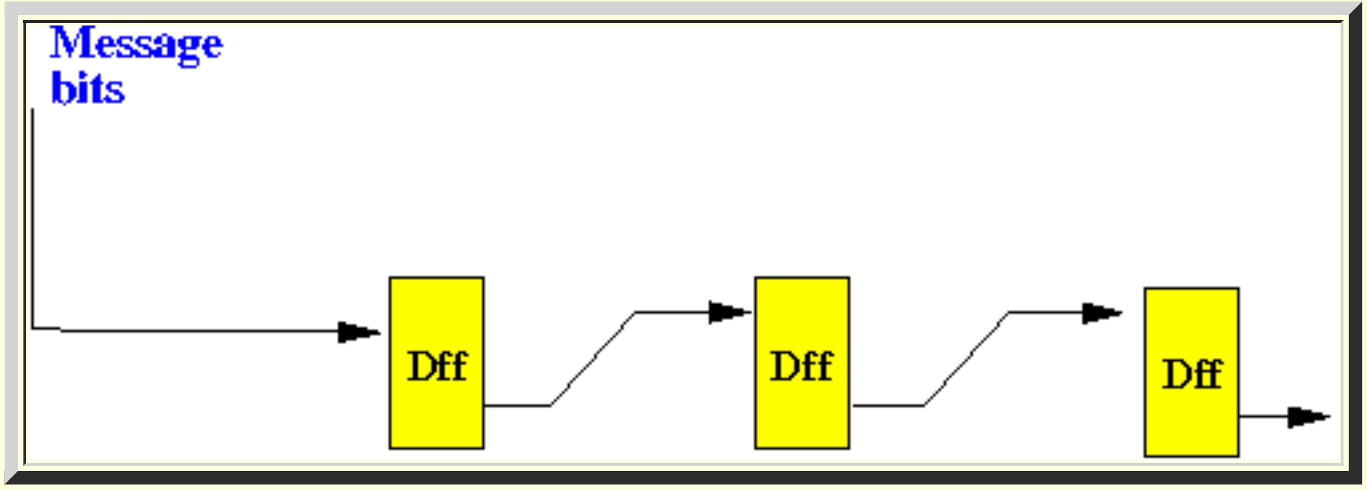
It's much easier to **see** and **example** (I'll do it in class)

- **Example:** **compute circuit** for **generator polynomial** **1001**

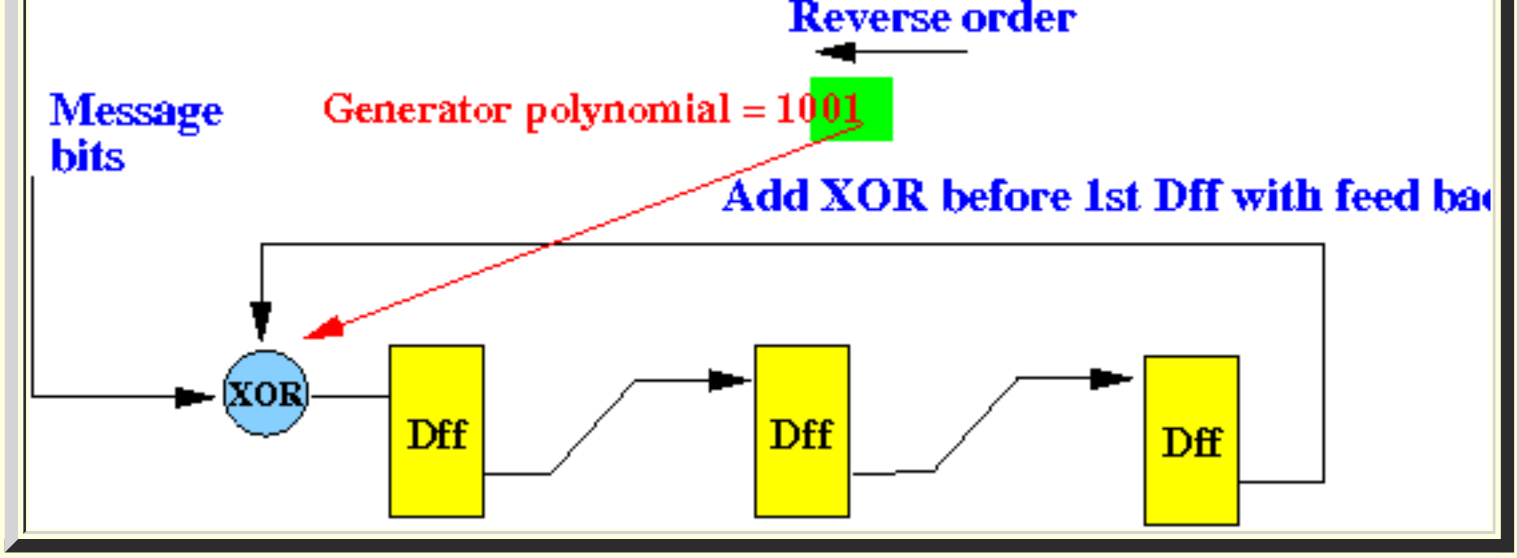
  - **N = 4**

    Put the **3 Dffs** in a row.

    Connect the **output** of the ***i^{th}*** **Dff** to the **input** of the ***i+1th*** **Dff**:

  - **Add** the **feed backs:**

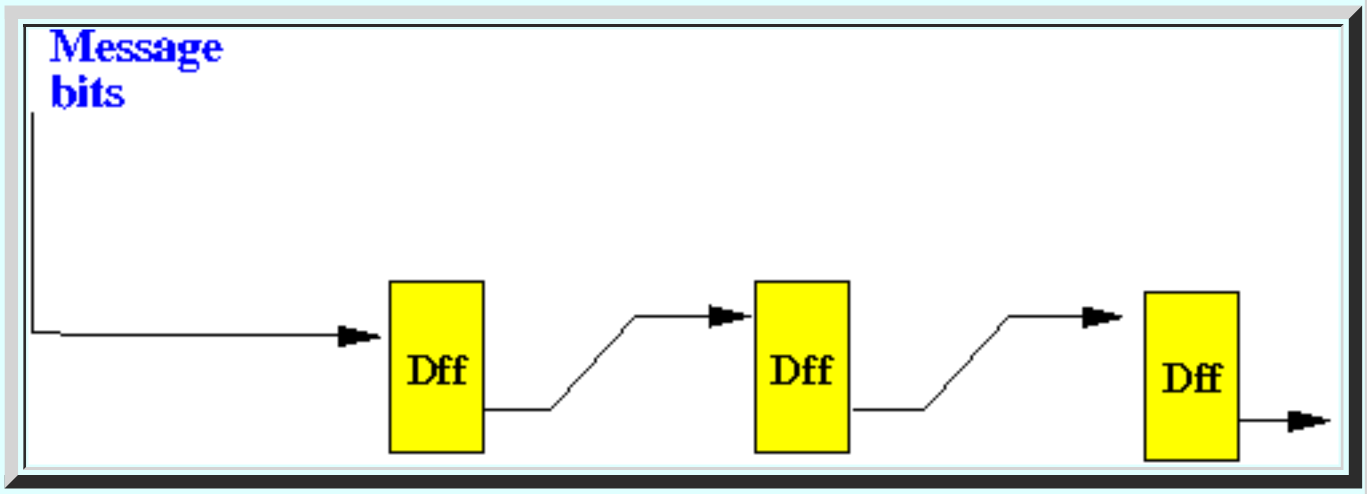


- **Example:** **compute circuit** for **generator polynomial** **1101**

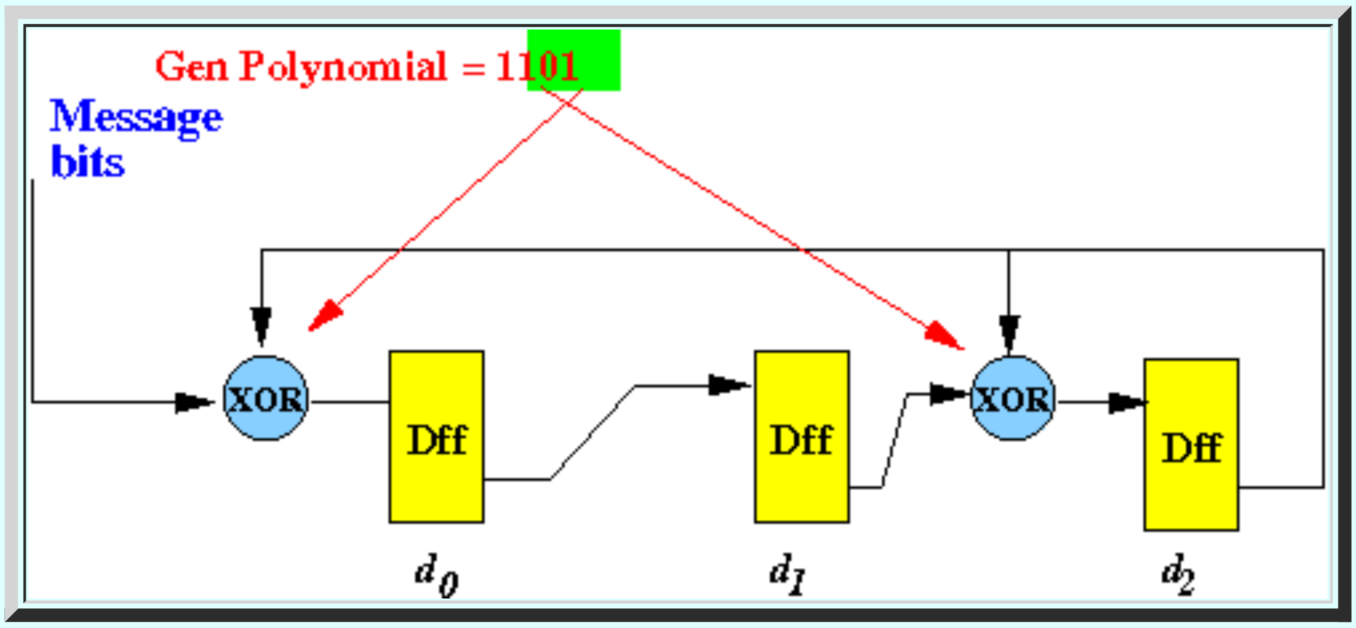
  - **N = 4**

    Put the **3 Dffs** in a row.

    Connect the **output** of the ***i*** th **Dff** to the **input** of the ***i+1*** th **Dff**:

  - **Add** the **feed backs:**

- **Computing the CRC with the hardware implementation**

  - **How to** compute the **CRC** for an **input message**:

- **Reset** all the **Dffs (memory elements)** to **ZERO**

    - Toggle key **'a'** to **ONE** to **clear** the **shift registers**
    - Toggle key **'a'** back to **ZERO (blank cell)** so you can **run** the **circuit**

- **Present** the **first bit** of the message

    - Use **key** **0** to set it to the **value** of the **first bit** (**0** or **1**)
    - The **input bits** are presented from **left to right**:

```
Message:    101010

Present the bits as follows:    1   0   1   0   1   0
```

- Give the **compute circuit** a **clock signal**

    - You must **toggle** the key **1** ***twice***

- Present the **second bit** of the message (see above)
- Give the **compute circuit** another **clock signal** (see above)
- And so on, until **all bits** in the message are processed

Note:

    - **Remember** that the **sender** must ***append*** ***N*–1** **ZERO (0)** bits to the **input message**
    - These ***N*–1** **ZERO (0)** bits must be **processed** ***also*** !!

- After processing **all bits**:

    - **Sender**:

        - **Remainder** of the **"division"** = **values** of the **shift registers** ***read*** in the ***reverse*** **order**

    - **Receiver**:

        - If **remainder** of the **"division"** = **all 0's**, then ***accept*** the **message**
        - **Otherwise**, ***reject*** the **message**

- **Example Program:** (Circuit for **generator polynomial** 1101)

Example

  - Prog file: click here

How to use it:

- Run: **/home/cs455001/demo/Logic-Sim/CRC1**
- Make **a = 0 (blank)** to run circuit
- Toggle **0:Input** to the next bit in input
- Press **1:Clk** to give circuit a clock signal (2 toggles per input bit).

This is a **logic-sim** program from my **CS355** class...

Example:

```
                 00011
               ----------------
         1101 /  10001000
                 1101
                 ---- (XOR)
                  1011
                  1101
                  ---- (XOR)
                   1100
                   1101
                   ---- (XOR)
                   000100 <---- remainder = 100
```

Do this:

- Toggle 2 to run circuit
- Toggle 0 (input) to **1**, then toggle 1 (clock **twice**)
- Toggle 0 (input) to **0**, then toggle 1 (clock **twice**)
- Toggle 0 (input) to **0**, then toggle 1 (clock **twice**)
- Toggle 0 (input) to **0**, then toggle 1 (clock **twice**)
- Toggle 0 (input) to **1**, then toggle 1 (clock **twice**)
- Toggle 0 (input) to **0**, then toggle 1 (clock **twice**)
- Toggle 0 (input) to **0**, then toggle 1 (clock **twice**)
- Toggle 0 (input) to **0**, then toggle 1 (clock **twice**)
- **Read** the bits **backwards**: **100**