August 2018

Manual of Pacemaker

**Author:** Han Wang

**Instructed by:** Dr. Minjun Seo, Prof. Fadi Kurdahi

Content

[Chapter 1 Pacemaker Hardware 2](#_Toc523830532)

[Chapter 2 Software Migration 5](#_Toc523830533)

[2.1 POSIX Threads 5](#_Toc523830534)

[Thread management: 5](#_Toc523830535)

[Mutexes: 5](#_Toc523830536)

[Condition Variables: 6](#_Toc523830537)

[Example of Synchronization 6](#_Toc523830538)

[2.2 Software Implementation 7](#_Toc523830539)

[VAI thread: 7](#_Toc523830540)

[Atrial thread: 9](#_Toc523830541)

[AVI thread: 10](#_Toc523830542)

[Ventricular thread: 11](#_Toc523830543)

[PVARP thread: 12](#_Toc523830544)

[VRP thread: 13](#_Toc523830545)

[Atrial Sensor thread: 14](#_Toc523830546)

[Atrial Sensor thread: 15](#_Toc523830547)

[Configuration thread: 16](#_Toc523830548)

[2.3 More comments 16](#_Toc523830549)

[Chapter 3 Communication 17](#_Toc523830550)

[3.1 Socket Programming 17](#_Toc523830551)

[Stage for server 17](#_Toc523830552)

[Stages for client 18](#_Toc523830553)

[Example 18](#_Toc523830554)

[3.2 Communication Protocol 18](#_Toc523830555)

[Predefined characters 18](#_Toc523830556)

[Token and data delivery method 19](#_Toc523830557)

[3.3 Communication Code 19](#_Toc523830558)

[Server: 19](#_Toc523830559)

[Client: 22](#_Toc523830560)

[3.4 Debugging on Board 25](#_Toc523830561)

[Bitsteam download 25](#_Toc523830562)

[JTAG with GRMON 26](#_Toc523830563)

[Chapter 4 MATLAB GUI 28](#_Toc523830564)

[4.1 Create a MATLAB GUI 28](#_Toc523830565)

[4.2 GUI of the Pacemaker 28](#_Toc523830566)

[Instruction 29](#_Toc523830567)

The basic module of pacemaker is established: atrial and ventricular pace could be generated by the pacemaker; Atrial-ventricular Interval (AVI) and Ventricular-atrial Interval (VAI) could be set by the outside MATLAB client; Post Ventricular Atrial Refractory (PVARP) and Ventricular Refractory Period (VRP) could be set inside the pacemaker code.

# Chapter 1 Pacemaker Hardware

The beginning of this pacemaker should be the understanding of the VHDL code of 6 pCores and their interactions with the original C code. It should be better explained with the structure shown below.

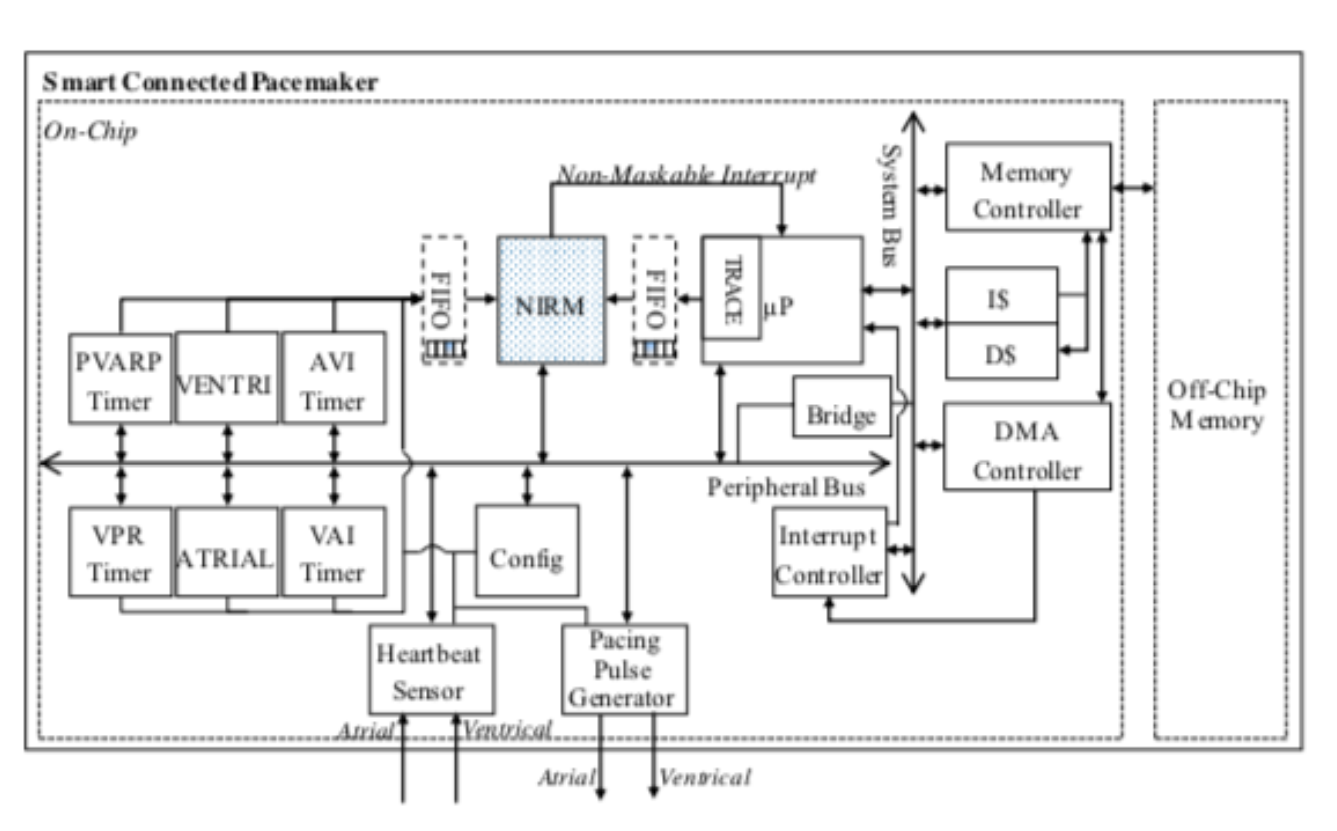


Fig. The structure of the pacemaker

The pcores of *VENTRI* and *ATRIAL* send signal to the *Pacing Pulse Generator* to generate Atrial and Ventricular pace; the other 4 pcores on the left side calculate the time determining the interval of the paces (*AVI Timer* and *VAI* *Timer*) and the interval during which the sensed pace or noise would be ignored (*PVARP* *Timer* and *VPR* *Timer*).

The example of *AVI* pcore is shown below. When the timer is enabled and triggered, it begins to calculate the time; when the calculation is done, it would trigger an interrupt and the software in C would handle this interrupt and trigger *VENTRI* pcore to generate ventricular pace.

AVI\_PORC: **process** (Bus2IP\_Clk) **is**

**begin**

**if** Bus2IP\_Clk'***event*** ***and*** Bus2IP\_Clk='1' **then**

**if** Bus2IP\_Resetn = '0' **then**

avi\_interr <='0';

avi\_counter <=0;

*--avi\_enable <='0';*

avi\_divider <=0;

**else**

**if** interr\_enable ='1' **then**

**if** avi\_trigger ='1' **then** *-- reset from sensor (include pace)*

*--avi\_enable <='1';*

avi\_counter <=0;

avi\_divider <=0;

**end** **if**;

**if** avi\_stop ='1' **then**

*--avi\_enable <='0';*

avi\_counter <=0;

avi\_divider <=0;

**else**

avi\_divider <=avi\_divider+1;

**end** **if**;

**if** avi\_counter = ***to\_integer***(***unsigned***(slv\_reg0)) **then**

avi\_counter <=0;

avi\_divider <=0;

*--avi\_enable <='0';*

avi\_interr <='1';

**end** **if**;

**if** avi\_divider =MILLSECOND **then**

avi\_divider <=0;

avi\_counter <=avi\_counter+1;

*--elsif avi\_enable ='1' then*

*-- avi\_divider <=avi\_divider+1;*

**end** **if**;

**if** avi\_interr ='1' **then**

avi\_interr <= '0'; *-- only interrupt once*

**end** **if**;

**end** **if**;

**end** **if**;

**end** **if**;

**end** **process** AVI\_PORC;

The basic working mode is presented in the sequence diagram of Fig. 2 and Fig. 3.

Fig. 2 shows the scenario when the patient’s heart could not generate natural pace, so the AVI and VAI timer would determine the heart rate of the patient. The atrial pace was generated in the Atrial pcore in which the AVI trigger and VRP trigger signal are sent. In the AVI pcore, it begins to calculate the time (150ms) after receiving the signal; in the VRP pcore, it begins to calculate the time (620ms), during which the VAI pcore could not send any signal to Atrial pcore. When the AVI timer is fired, it trigger the Ventri pcore to generate the ventricular pace, and then Ventri pcore would send VAI and PVARP trigger signal. In the VAI pcore, the VAI time (850ms) begins to be calculated and in the PVARP, the filter time (350ms) begins to be calculated during which the AVI pcore could not send any signal to Ventri pcore. When the VAI timer is fired, it trigger the Atrial pcore and above presents the loop of this pacemaker.

Fig. 3 shows the cases when the patient could internally generate the cardiac pace. If the sensor detected the atrial or ventricular pace generated by the patient himself, it would stop the VAI or VAI timer and would not generate pace in the Atrial or Ventri pcores. It would directly go to the next step, triggering the AVI or VAI timer and the filter.



Fig. Sequence diagram of pacemaker, no pace sensed



Fig. Sequence diagram of pacemaker, pace sensed

# Chapter 2 Software Migration

## 2.1 POSIX Threads

For our implementation, POSIX Threads (ptherads) are used to immigrate 6 pcores into software. POSIX Threads, usually referred to as pthreads, is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API.

Pthreads defines a set of C programming language types, functions and constants. It is implemented with a ***pthread.h*** header and a thread library. The functions of thread management, mutual exclusions (Mutexes) and conditional variables are used in the pacemaker. The brief introduction of these are as follows.

### Thread management:

**pthread\_create (thread,attr,start\_routine,arg):**

* **pthread\_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.
* **pthread\_create** arguments:
  + thread: An opaque, unique identifier for the new thread returned by the subroutine.
  + attr: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
  + start\_routine: the C routine that the thread will execute once it is created.
  + arg: A single argument that may be passed to start\_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

**pthread\_join (threadid,status):**

* The **pthread\_join()** subroutine blocks the calling thread until the specified **threadid** thread terminates.

### Mutexes:

* Mutex variables must be declared with type pthread\_mutex\_t, and must be initialized before they can be used. There are two ways to initialize a mutex variable:

1. Statically, when it is declared. For example:

pthread\_mutex\_t mymutex = PTHREAD\_MUTEX\_INITIALIZER;

1. Dynamically, with the pthread\_mutex\_init() routine. This method permits setting mutex object attributes, attr.

The mutex is initially unlocked.

Functions:

**pthread\_mutex\_lock (mutex)**

**pthread\_mutex\_unlock (mutex)**

* The **pthread\_mutex\_lock()** routine is used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
* **pthread\_mutex\_unlock()** will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. An error will be returned if:
* If the mutex was already unlocked
* If the mutex is owned by another thread

### Condition Variables:

* Condition variables must be declared with type pthread\_cond\_t, and must be initialized before they can be used. There are two ways to initialize a condition variable:

1. Statically, when it is declared. For example:

pthread\_cond\_t myconvar = PTHREAD\_COND\_INITIALIZER;

1. Dynamically, with the pthread\_cond\_init() routine. The ID of the created condition variable is returned to the calling thread through the condition parameter. This method permits setting condition variable object attributes, attr.

Functions:

**pthread\_cond\_wait (condition,mutex)**

**pthread\_cond\_signal (condition)**

**pthread\_cond\_broadcast (condition)**

* **pthread\_cond\_wait()** blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, mutex will be automatically locked for use by the thread. The programmer is then responsible for unlocking mutex when the thread is finished with it.

**Recommendation**: Using a WHILE loop instead of an IF statement (see watch\_count routine in example below) to check the waited for condition can help deal with several potential problems, such as:

* If several threads are waiting for the same wake up signal, they will take turns acquiring the mutex, and any one of them can then modify the condition they all waited for.
* If the thread received the signal in error due to a program bug
* The Pthreads library is permitted to issue spurious wake ups to a waiting thread without violating the standard.
* The **pthread\_cond\_signal()** routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for **pthread\_cond\_wait()** routine to complete.
* The **pthread\_cond\_broadcast()** routine should be used instead of **pthread\_cond\_signal()** if more than one thread is in a blocking wait state.
* It is a logical error to call **pthread\_cond\_signal()** before calling **pthread\_cond\_wait()**.

### Example of Synchronization

The code and brief explanation are shown below:

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <pthread.h>

**#include** <time.h>

pthread\_mutex\_t lock\_start = PTHREAD\_MUTEX\_INITIALIZER ;

pthread\_cond\_t cv\_start = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t lock1 = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t cv1 = PTHREAD\_COND\_INITIALIZER;

pthread\_t id1 , id2 ;

**void**\* **th1** ( **void**\* arg ) {

**while**(1)

{

pthread\_mutex\_lock(&lock1);

pthread\_cond\_wait(&cv1, &lock1);

pthread\_mutex\_unlock(&lock1);

**long** time\_after\_boot = rtems\_clock\_get\_ticks\_since\_boot();

printf("th1!!, interval= %ld ms \n",time\_after\_boot);

sleep(1);

}

pthread\_exit (NULL) ;

}

**void**\* **th2** ( **void**\* arg ) {

**while**(1)

{

pthread\_mutex\_lock(&lock1);

printf("th2!!\n");

pthread\_cond\_broadcast(&cv1);

pthread\_mutex\_unlock(&lock1);

sleep(2);

}

pthread\_exit (NULL ) ;

}

**void**\* **POSIX\_Init** ( **void** \*argument ) {

**if** (pthread\_create (&id1 ,NULL, th1 ,NULL) != 0 )

printf ( " pthread\_create1 " ) ;

**if** (pthread\_create (&id2 ,NULL, th2 ,NULL) != 0 )

printf ( " pthread\_create2 " ) ;

sleep(1);

**if** (pthread\_join ( id1 ,NULL) != 0 )

printf ( " pthread\_join 1 " ) ;

**if** (pthread\_join ( id2 ,NULL) != 0 )

printf ( " pthread\_join 2 " ) ;

printf ( " End of the a p p l i c a t i o n \n " ) ;

exit ( 0 ) ;

}

Result:

The interval in *th1* (1 second) is much shorter than that in *th2* (2 seconds), but because of the mutex lock and the conditional variable, the *printf()* in *th2* is executed first followed by *printf()* in *th1*.

## 2.2 Software Implementation

With the utilization of pthreads, each pcores could be transplanted into independent thread. The code and explanation of each thread are as follows.

### VAI thread:

Code:

**void** \***inc\_VAI**(**void** \*x\_void\_ptr)

{

**struct** timespec LocalTime\_VAI;

**struct** timespec outtime;

**struct** timeval now;

pthread\_mutex\_lock(&lock\_start);

pthread\_cond\_wait(&cv\_start, &lock\_start);

printf("inc\_VAI!!\n");

pthread\_mutex\_unlock(&lock\_start);

**while**(1)

{

/\* wait for the VRP filter \*/

**if**(VAI\_init==1) VAI\_init=0;

**else** **if**(flag\_exit)

{

pthread\_mutex\_lock(&lock\_VAI);

pthread\_cond\_broadcast(&cv\_VAI);

pthread\_mutex\_unlock(&lock\_VAI);

printf("VAI last send /n");

Is\_Going\_To\_Atrial=1;

Is\_Going\_To\_Ventri=1;

**break**;

}

**else**

{

pthread\_mutex\_lock(&lock\_VRP);

pthread\_cond\_wait(&cv\_VRP, &lock\_VRP);

pthread\_mutex\_unlock(&lock\_VRP);

}

/\* send message to Atrial using pthread\_cond\_timedwait() \*/

time\_counter++; //time\_counter==1

heart\_rate=60.0/(((**float**)time\_rate1+(**float**)time\_rate2)/1000.0);

**if**(heart\_rate<100)

{

heart\_rate=heart\_rate \* 10;

}

//printf("heart rate = %d ",heart\_rate);

**if**(request==1 && token==3)

{

puts("token = 3 here");

//here send haeat\_rate to matlab

G\_heart=heart\_rate;

pthread\_mutex\_lock(&lock\_G\_heart);

pthread\_cond\_broadcast(&cv\_G\_heart);

pthread\_mutex\_unlock(&lock\_G\_heart);

token=0;

}

//printf("VAI timer begin!\n");

gettimeofday(&now, NULL);

**if**((now.tv\_usec\* 1000+VAI\_ms\*1000000)/1000000000) outtime.tv\_sec = now.tv\_sec+VAI\_s+1 ;

**else** outtime.tv\_sec = now.tv\_sec+VAI\_s;

outtime.tv\_nsec =(now.tv\_usec\* 1000+VAI\_ms\*1000000)%1000000000;

pthread\_mutex\_lock(&lock\_time\_VAI);

pthread\_cond\_timedwait(&cv\_time\_VAI, &lock\_time\_VAI, &outtime); //850ms

pthread\_mutex\_unlock(&lock\_time\_VAI);

pthread\_mutex\_lock(&lock\_VAI);

pthread\_cond\_broadcast(&cv\_VAI);

//printf("VAI signal sent!!\n");

pthread\_mutex\_unlock(&lock\_VAI);

/\*wait for VAI trig\*/

pthread\_mutex\_lock(&lock\_VAI\_trig);

pthread\_cond\_wait(&cv\_VAI\_trig, &lock\_VAI\_trig);

pthread\_mutex\_unlock(&lock\_VAI\_trig);

Is\_Going\_To\_Ventri=0; //Reset sensor,Enable sensor!

}

printf("t\_VAI exit\n");

**return** 0;

}

Explanation:

1. After the establishment of the thread, a while() loop begins.
2. When entering the loop for the first time, because VRP filter has not been triggered, the program would skip the waiting for the VRP and go to the next step directly, but after the second time when the loop begins, it would wait for the VRP signal.
3. Then it would calculate the heart rate, using the variables which record the AVI and VAI time.
4. If it should give the heart rate data to the host PC, it would send signal to the communication thread (the communication part would be specified later).
5. Then the VAI timer begins. The use of *pthread\_cond\_timedwait()* means that if the atrial pace are sensed internally, the timer would immediately stop and if not, it would wait for an VAI time.
6. After the VAI time, it would send the signal to Atrial thread to take some action.
7. And it would wait for a signal from the Ventricular thread to start the loop again.

### Atrial thread:

Code:

**void** \***inc\_ATRIAL**(**void** \*x\_void\_ptr)

{

**int** VAItime;

**struct** timespec LocalTime\_AVI;

pthread\_mutex\_lock(&lock\_start);

printf("inc\_ATRIAL!!\n");

pthread\_cond\_wait(&cv\_start, &lock\_start);

pthread\_mutex\_unlock(&lock\_start);

**while**(1)

{

/\*wait for VAI signal \*/

pthread\_mutex\_lock(&lock\_VAI);

pthread\_cond\_wait(&cv\_VAI, &lock\_VAI);

pthread\_mutex\_unlock(&lock\_VAI);

Is\_Going\_To\_Atrial=1; //Sensor disable!

**if**(flag\_exit)

{

pthread\_mutex\_lock(&lock\_AVI\_trig);

pthread\_cond\_broadcast(&cv\_AVI\_trig);

//printf("AVI trigger signal sent, last one!!\n");

pthread\_mutex\_unlock(&lock\_AVI\_trig);

Is\_Going\_To\_Atrial=1;

Is\_Going\_To\_Ventri=1;

**break**;

}

time\_after\_boot[time\_counter]=rtems\_clock\_get\_ticks\_since\_boot();

**long** VAItime2=(time\_after\_boot[time\_counter]-time\_after\_boot[time\_counter-1])\*10;

**if**(flag\_AS==0)

printf("atrial pace generated!, VAI time interval = %d ms \n",VAItime2);

**else**

flag\_AS=0;

**if** (request==1 && token==1) //send the VAI time if needed

{

G\_VAI=VAItime2;

time\_rate1=VAItime2;

puts("token == 1 here");

**if**(G\_VAI<1000)

{

G\_VAI+=9000;

}

pthread\_mutex\_lock(&lock\_G\_VAI);

pthread\_cond\_broadcast(&cv\_G\_VAI);

pthread\_mutex\_unlock(&lock\_G\_VAI);

token=2;

}

time\_counter++; //time\_counter==2

/\*send message to trig AVI timer and VRP filter\*/

pthread\_mAutex\_lock(&lock\_AVI\_trig);

pthread\_cond\_broadcast(&cv\_AVI\_trig);

//printf("AVI trig signal sent!!\n");

pthread\_mutex\_unlock(&lock\_AVI\_trig);

}

printf("t\_ATRIAL exit!\n");

**return** 0;

}

Explanation:

1. After the establishment of the thread, a while() loop begins.
2. Then the program would wait for the broadcast signal from VAI thread and then determines if there is an atrial pace sensed. If not, it would generate an atrial pace.
3. It would send the VAI time data if needed.
4. After the pace is generated, this thread would send the broadcast signal to the AVI thread and VRP thread.

### AVI thread:

Code:

**void** \***inc\_AVI**(**void** \*x\_void\_ptr)

{

**long** AVItime;

**struct** timespec LocalTime\_AVI;

**struct** timespec outtime;

**struct** timeval now;

pthread\_mutex\_lock(&lock\_start);

pthread\_cond\_wait(&cv\_start, &lock\_start);

printf("inc\_AVI!!\n");

pthread\_mutex\_unlock(&lock\_start);

**while**(1)

{

/\* Wait for PVARP filter \*/

**if**(AVI\_init==1) AVI\_init=0;

**else** **if**(flag\_exit)

{

pthread\_mutex\_lock(&lock\_AVI);

pthread\_cond\_broadcast(&cv\_AVI);

printf("AVI timer trigger next to end!\n");

pthread\_mutex\_unlock(&lock\_AVI);

Is\_Going\_To\_Atrial=1;

Is\_Going\_To\_Ventri=1;

**break**;

}

**else**

{

// printf("wait for the PVARP \n");

pthread\_mutex\_lock(&lock\_PVARP);

pthread\_cond\_wait(&cv\_PVARP, &lock\_PVARP);

pthread\_mutex\_unlock(&lock\_PVARP);

}

/\*wait for the AVI trigger\*/

pthread\_mutex\_lock(&lock\_AVI\_trig);

pthread\_cond\_wait(&cv\_AVI\_trig, &lock\_AVI\_trig);

pthread\_mutex\_unlock(&lock\_AVI\_trig);

Is\_Going\_To\_Atrial=0; //Reset flag, Enable Sensor

**if**(flag\_exit)

{

pthread\_mutex\_lock(&lock\_AVI);

pthread\_cond\_broadcast(&cv\_AVI);

//printf("AVI timer trigger next to end!\n");

pthread\_mutex\_unlock(&lock\_AVI);

Is\_Going\_To\_Atrial=1;

Is\_Going\_To\_Ventri=1;

**break**;

}

/\*AVI timer begin\*/

time\_after\_boot[time\_counter]=rtems\_clock\_get\_ticks\_since\_boot(); //me\_counter==2

//printf("AVI timer begin! before AVI begin delay = %d ms\n",AVI\_delay);

time\_counter++; //time\_counter==3

gettimeofday(&now, NULL);

**if**((now.tv\_usec\* 1000+AVI\_ms\*1000000)/1000000000==1) outtime.tv\_sec = now.tv\_sec+AVI\_s+1;

**else** outtime.tv\_sec = now.tv\_sec+AVI\_s;

outtime.tv\_nsec =(now.tv\_usec\* 1000+AVI\_ms\*1000000)%1000000000;

pthread\_mutex\_lock(&lock\_time\_AVI);

pthread\_cond\_timedwait(&cv\_time\_AVI, &lock\_time\_AVI, &outtime);//150ms

pthread\_mutex\_unlock(&lock\_time\_AVI);

pthread\_mutex\_lock(&lock\_AVI);

pthread\_cond\_broadcast(&cv\_AVI);

time\_after\_boot[time\_counter]=rtems\_clock\_get\_ticks\_since\_boot();//time\_counter==3

AVItime=(time\_after\_boot[time\_counter]-time\_after\_boot[time\_counter-1])\*10;

//printf("AVI signal sent!! AVI time interval= %d ms \n",AVItime );

pthread\_mutex\_unlock(&lock\_AVI);

**if** (request==1 && token==2) //send to the Matlab if needed

{

G\_AVI=AVItime;

time\_rate2=AVItime;

**if**(G\_AVI<100)

G\_AVI+=900;

puts("token = 2 here !");

pthread\_mutex\_lock(&lock\_G\_AVI);

pthread\_cond\_broadcast(&cv\_G\_AVI);

pthread\_mutex\_unlock(&lock\_G\_AVI);

token=3;

}

time\_counter++; //time\_counter==4

}

printf("t\_AVI exit\n");

**return** 0;

}

Explanation:

1. After the establishment of the thread, a while() loop begins.
2. When entering the loop for the first time, because PVARP filter has not been triggered, the program would skip the waiting for the PVARP and go to the next step directly, but after the second time when the loop begins, it would wait for the PVARP signal.
3. Then it waits for the beginning of the AVI timer until it receive the broadcast signal from the Atrial thread.
4. The AVI timer use the function of *pthread\_cond\_timedwait()*,meaning that if the ventricular pace are sensed internally, the timer would immediately stop and if not, it would wait for an AVI time.
5. After the VAI time, it would send the signal to Atrial thread to take some action.
6. Then it calculate the AVI time and send to the Matlab side if needed.

### Ventricular thread:

Code:

**void** \***inc\_VENTRI**(**void** \*x\_void\_ptr)

{

**struct** timespec LocalTime;

pthread\_mutex\_lock(&lock\_start);

printf("inc\_VENTRI!!\n");

pthread\_cond\_wait(&cv\_start, &lock\_start);

pthread\_mutex\_unlock(&lock\_start);

**while**(1)

{

/\*wait for AVI signal\*/

pthread\_mutex\_lock(&lock\_AVI);

pthread\_cond\_wait(&cv\_AVI, &lock\_AVI);

pthread\_mutex\_unlock(&lock\_AVI);

Is\_Going\_To\_Ventri=1;

**if**(flag\_exit)

{

pthread\_mutex\_lock(&lock\_VAI\_trig);

pthread\_cond\_broadcast(&cv\_VAI\_trig);

printf("Ventric trigger VIA to end\n");

pthread\_mutex\_unlock(&lock\_VAI\_trig);

Is\_Going\_To\_Atrial=1;

Is\_Going\_To\_Ventri=1;

**break**;

}

time\_after\_boot[time\_counter]=rtems\_clock\_get\_ticks\_since\_boot();//time\_counter==4

**if**(flag\_VS==0)

printf("ventricular pace generated!\n");

**else**

flag\_VS=0;

time\_counter=0; //time\_counter==0

/\*send trigger VAI timer To VAI & PVARP \*/

pthread\_mutex\_lock(&lock\_VAI\_trig);

pthread\_cond\_broadcast(&cv\_VAI\_trig);

//printf("VAI trig signal sent!!\n");

pthread\_mutex\_unlock(&lock\_VAI\_trig);

}

printf("t\_VENTRI exit!\n");

**return** 0;

}

Explanation:

1. After the establishment of the thread, a while() loop begins.
2. Then the program would wait for the broadcast signal from AVI thread and then determines if there is a ventricular pace sensed. If not, it would generate a ventricular pace.
3. After the pace is generated, this thread would send the broadcast signal to the VAI thread and PVARP thread.

### PVARP thread:

Code:

**void** \***inc\_PVARP**(**void** \*x\_void\_ptr)

{

**struct** timespec LocalTime;

pthread\_mutex\_lock(&lock\_start);

pthread\_cond\_wait(&cv\_start, &lock\_start);

pthread\_mutex\_unlock(&lock\_start);

printf("inc\_PVARP!!\n");

**while**(1)

{

/\* Wait for the VAI trigger\*/

pthread\_mutex\_lock(&lock\_VAI\_trig);

pthread\_cond\_wait(&cv\_VAI\_trig, &lock\_VAI\_trig);

pthread\_mutex\_unlock(&lock\_VAI\_trig);

**if**(flag\_exit)

{

pthread\_mutex\_lock(&lock\_PVARP);

pthread\_cond\_broadcast(&cv\_PVARP);

pthread\_mutex\_unlock(&lock\_PVARP);

Is\_Going\_To\_Atrial=1;

Is\_Going\_To\_Ventri=1;

**break**;

}

/\* Start PVARP filter\*/

LocalTime.tv\_sec = PVARP\_s;

LocalTime.tv\_nsec = PVARP\_ms\*1000000; //ms

//printf("PVARP filter timer START! \n");

flag\_PVARP=1;

nanosleep(&LocalTime,NULL);

flag\_PVARP=0;

//printf("PVARP Filter timer runs out!!\n");

pthread\_mutex\_lock(&lock\_PVARP);

pthread\_cond\_broadcast(&cv\_PVARP);

pthread\_mutex\_unlock(&lock\_PVARP);

**if**(flag\_exit)

{

Is\_Going\_To\_Atrial=1; //reset the flag, facilitate sensed

Is\_Going\_To\_Ventri=1; // thread to exit

**break**;

}

}

printf("t\_PVARP exit\n");

**return** 0;

}

Explanation:

1. After the establishment of the thread, a while() loop begins.
2. The PVARP timer begins when it receives the broadcast signal from the Ventricular thread. A flag would be set to mark that at this time, sensor’s data should be ignored.
3. When the timer is fired, it would reset the flag and send the signal to the AVI thread, to enable the next step.

### VRP thread:

Code:

**void** \***inc\_VRP**(**void** \*x\_void\_ptr)

{

**struct** timespec LocalTime;

pthread\_mutex\_lock(&lock\_start);

pthread\_cond\_wait(&cv\_start, &lock\_start);

printf("inc\_VRP!!\n");

pthread\_mutex\_unlock(&lock\_start);

**while**(1)

{

/\*wait for the AVI trigger\*/

pthread\_mutex\_lock(&lock\_AVI\_trig);

pthread\_cond\_wait(&cv\_AVI\_trig, &lock\_AVI\_trig);

pthread\_mutex\_unlock(&lock\_AVI\_trig);

**if**(flag\_exit)

{

pthread\_mutex\_lock(&lock\_VRP);

pthread\_cond\_broadcast(&cv\_VRP);

pthread\_mutex\_unlock(&lock\_VRP);

Is\_Going\_To\_Atrial=1;

Is\_Going\_To\_Ventri=1;

**break**;

}

/\*start VRP filter\*/

LocalTime.tv\_sec = VRP\_s;

LocalTime.tv\_nsec = VRP\_ms\*1000000; //ms

//printf("VRP filter timer START! \n");

flag\_VRP=1;

nanosleep(&LocalTime,NULL);

flag\_VRP=0;

//printf("VRP Filter timer runs out!!\n");

pthread\_mutex\_lock(&lock\_VRP);

pthread\_cond\_broadcast(&cv\_VRP);

pthread\_mutex\_unlock(&lock\_VRP);

**if**(flag\_exit)

{

Is\_Going\_To\_Atrial=1;

Is\_Going\_To\_Ventri=1;

**break**;

}

}

printf("t\_VRP exit\n");

**return** 0;

}

Explanation:

1. After the establishment of the thread, a while() loop begins.
2. The VRP timer begins when it receives the broadcast signal from the Atrial thread. A flag would be set to mark that at this time, sensor’s data should be ignored.
3. When the timer is fired, it would reset the flag and send the signal to the VAI thread, to enable the next step.

### Atrial Sensor thread:

Code:

**void** \***inc\_Sensed\_Atrial**(**void** \*x\_void\_ptr)

{

**struct** timespec LocalTime;

pthread\_mutex\_lock(&lock\_start);

pthread\_cond\_wait(&cv\_start, &lock\_start);

printf("inc\_Sensed\_ATRIAL!!\n");

pthread\_mutex\_unlock(&lock\_start);

**while**(1)

{

/\*wait for VAI trig\*/

pthread\_mutex\_lock(&lock\_VAI\_trig);

pthread\_cond\_wait(&cv\_VAI\_trig, &lock\_VAI\_trig);

pthread\_mutex\_unlock(&lock\_VAI\_trig);

**while**(1)

{

LocalTime.tv\_sec = 0;

LocalTime.tv\_nsec = 5\*1000000; //5ms

nanosleep(&LocalTime,NULL);

**if**(check\_buffer(ATRIAL\_BUFF))

{

**if**(flag\_VRP)

**continue**; //if the filter is going on, than ignore the sensed data

flag\_AS=1;

pthread\_cond\_broadcast(&cv\_time\_VAI);

pthread\_mutex\_unlock(&lock\_time\_VAI);

**break**;

}

**if**(Is\_Going\_To\_Atrial)

{

Is\_Going\_To\_Atrial=0;

**break**;

}

}

**if**(flag\_exit)

{

Is\_Going\_To\_Atrial=1;

Is\_Going\_To\_Ventri=1;

**break**;

}

}

printf("inc\_Sensed\_Atrial exit\n");

**return** 0;

}

Explanation:

1. After the establishment of the thread, a while() loop begins.
2. After receiving the signal from Ventricular thread, it goes into a loop, checking the atrial sensor buffer every 5 millisecond.
3. If the atrial pace is not sensed, then the program would goes into Atrial thread and generate a pace, the flag Is\_Going\_To\_Atrial would be set and the inner loop would be terminated.
4. If the atrial pace is sensed when the filter is terminated, it would send a signal to VAI thread to stop VAI timer and a flag of Atrial sensed would be set.

### Atrial Sensor thread:

Code:

**void** \***inc\_Sensed\_Ventri**(**void** \*x\_void\_ptr)

{

**struct** timespec LocalTime;

pthread\_mutex\_lock(&lock\_start);

pthread\_cond\_wait(&cv\_start, &lock\_start);

printf("inc\_Sensed\_Ventri!!\n");

pthread\_mutex\_unlock(&lock\_start);

**while**(1)

{

/\*wait for AVI trig\*/

pthread\_mutex\_lock(&lock\_AVI\_trig);

pthread\_cond\_wait(&cv\_VAI\_trig, &lock\_AVI\_trig);

pthread\_mutex\_unlock(&lock\_AVI\_trig);

**while**(1)

{

LocalTime.tv\_sec = 0;

LocalTime.tv\_nsec = 5\*1000000; //5ms

nanosleep(&LocalTime,NULL);

**if**(check\_buffer(VENTRI\_BUFF))

{

**if**(flag\_PVARP)

**continue**;

flag\_VS=1;

pthread\_cond\_broadcast(&cv\_time\_AVI);

pthread\_mutex\_unlock(&lock\_time\_AVI);

**break**;

}

**if**(Is\_Going\_To\_Ventri)

{

Is\_Going\_To\_Ventri=0;

**break**;

}

}

**if**(flag\_exit)

{

Is\_Going\_To\_Atrial=1;

Is\_Going\_To\_Ventri=1;

**break**;

}

}

printf("inc\_Sensed\_Ventri exit\n");

**return** 0;

}

Explanation:

1. After the establishment of the thread, a while() loop begins.
2. After receiving the signal from Atrial thread, it goes into a loop, checking the atrial sensor buffer every 5 millisecond.
3. If the ventricular pace is not sensed, then the program would goes into Ventricular thread and generate a pace, the flag Is\_Going\_To\_Ventri would be set and the inner loop would be terminated.
4. If the atrial pace is sensed when the filter is terminated, it would send a signal to AVI thread to stop AVI timer and a flag of Ventricular sensed would be set.

### Configuration thread:

Code:

**void** \***inc\_config**(**void** \*x\_void\_ptr)

{

**struct** timespec outtime;

**struct** timeval now;

pthread\_mutex\_lock(&lock\_start);

pthread\_cond\_wait(&cv\_start, &lock\_start);

printf("inc\_config!!\n");

pthread\_mutex\_unlock(&lock\_start);

**while**(1)

{

outtime.tv\_sec = 0;

outtime.tv\_nsec = doc\_check\_interval\*1000000; //ms

nanosleep(&outtime,NULL);

**if**(doc\_init==1)

{

flag\_exit=1;

**break**;

}

}

count++;

printf("t\_config exit");

**return** 0;

}

Explanation:

1. After the establishment of the thread, a while() loop begins.
2. This thread could set the flag of exit to kill all the threads except the communication thread.

## 2.3 More comments

* Exit strategy: the variable of flag\_exit is the exit sign of each treads which would be checked at least one time in the loop of each thread. If this flag is set in the configuration thread, the loops in the other threads would be broken while at the same time, each loop would send broadcast signals to the associated threads to guarantee all the mutexes could unlock. Besides, the variables of Is\_Going\_To\_Atrial and Is\_Going\_To\_Ventri would be set so that the inner loops of the two sensor threads would be broken and the threads could exit. This exit strategy guarantees all thread except the communication thread could exit in a good order and mutex locks and conditional variables could be initialized.
* The reason why there are always a nanosleep() or sleep() in a loop where these function seem unnecessary is that it would prevent a block of other thread. Although the pthreads are supposed to work in parallel, they would be blocked if other thread takes all resources.
* The communication thread is not shown in these chapter which would be specified in the chapter 3 with the communication strategy.

# Chapter 3 Communication

In this chapter, the communication strategy is introduced and specified. The pacemakers could communicate with the PC using Ethernet and socket programming. It could receive different commands from PC, AVI, VAI time could be set, threads could be initialized and it could send the AVI, VAI, heart rate data to the PC.

## 3.1 Socket Programming

Socket Programming is a way to establish a connection between two devices on a network. A particular port at an IP would be listened by one socket while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.

The Fig. 4 shows the state diagram for server-client socket module.

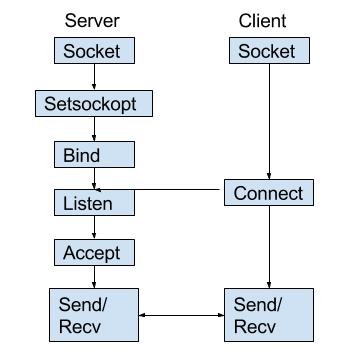


Fig. Server-client module

### Stage for server

1. Socket creation:

int sockfd = socket(domain, type, protocol);

**sockfd:** socket descriptor, an integer (like a file-handle)

**domain:** integer, communication domain e.g., AF\_INET (IPv4 protocol) , AF\_INET6 (IPv6 protocol)

**type:** communication type

SOCK\_STREAM: TCP(reliable, connection oriented)

SOCK\_DGRAM: UDP(unreliable, connectionless)

**protocol:**Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet. (man protocols for more details)

1. Setsockopt:

int setsockopt(int sockfd, int level, int optname,

const void \*optval, socklen\_t optlen);

This helps in manipulating options for the socket referred by the file descriptor sockfd. This is completely optional, but it helps in reuse of address and port. Prevents error such as: “address already in use”.

1. Bind:

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR\_ANY to specify the IP address.

1. Listen:

int listen(int sockfd, int backlog);

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

1. Accept:

int new\_socket= accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

### Stages for client

1. Socket connection:

Exactly same as that of server’s socket creation.

1. Connect:

int connect(int sockfd, const struct sockaddr \*addr,

socklen\_t addrlen);

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server’s address and port is specified in addr.

### Example

Here the server-client socket example is presented

server.c:

client.c:

## 3.2 Communication Protocol

To achieve an effective and efficient communication between the pacemaker and the PC, a protocol is designed. In this application, the on-board pacemaker would be a server while the PC a client. Several commands are predefined and a token is used when the pacemaker deliver the AVI, VAI and heart rate data to the client, making the transmission of the data flow a good order.

### Predefined characters

After the establishment of the socket, the server would wait for the connection of the client. When they are connected, the server would receive the data from the client. The first character it gets from the client would be judged to see what the next step is. The means of different characters is predefined:

1. Character ‘0’: to stop the pacemaker.
2. Character ‘1’: to start the pacemaker.
3. Character ‘2’: to disconnect the pacemaker
4. Character ‘3’: to get history data
5. Character ‘8’: to set the AVI and VAI value
6. Character ‘9’: to request AVI, VAI and heart rate data from the pacemaker.

### Token and data delivery method

To make the data flow in a good order, and to guarantee that the actions of sending and receiving would not happen at the same time, a token is used. Four thread would have the chance to hold the token: communication thread, Atrial thread, AVI thread and VAI thread. When a thread holds the token, it will get the opportunity to send or receive the data.

* When the token equals to ‘0’, the communication thread would get chance to receive the data from the client.
* When the token equals to ‘1’, the Atrial thread would get chance to send VAI data.
* When the token equals to ‘2’, the AVI thread would get chance to send AVI data.
* When the token equals to ‘3’, the VAI thread would get chance to send heart rate data.

When the data is about to send from pacemaker, several scenarios must be considered. Since the digits of each data may vary: the AVI time may be 120ms (3 digits) or 90ms (2 digits), the VAI time may be three or four digits and heart rate may be two or three digits. For each data to be delivered, the length should be regulated.

* Four-digit length is set for VAI data: if the length of VAI data is thee digits, such as 850ms, it would be sent as 9850 (9000+850), since it is impossible for the human being to have such long VAI. If the client receive the data value which is more than 9000, it would regard this as the result after added by 9000, otherwise it would regard this as an original VAI.
* Three-digit length is set for AVI data: if the length of AVI data is two digits, such as 90ms, it would be sent as 990 (900+90), since it is impossible for the human being to have such long AVI. If the client receive the data value which is more than 900, it would regard this as the result after added by 900, otherwise it would regard this as an original AVI.
* Three-digit length is set for heart rate data: if the length of heart rate data is two digits, such as 60 beats/min, it would be sent as 600 (60\*10), and if the client receive the data value which is more than 300, it would regard this as the result after multiplied by 10, otherwise it would regard this as an original heart rate.

From the client side, if the bottom of data request is clicked, it would step into a loop where the character of ‘9’ would be sent in the first place, and then the client would receive three separate data. So it would maintain a data flow transmission. When the stop character ‘0’ is sent, the pacemaker would stop generating pace and all thread would be killed except the communication thread.

## 3.3 Communication Code

To implement communication between two devices, the C code and Matlab code is presented and explained.

### Server:

Code:

**void**\* **th1** ( **void**\* arg ) {

**struct** timespec outtime;

**char** str9[3]="9",str0[3]="0",str8[3]="8",str1[3]="1",str2[3]="2",msg\_back[4];

**int** socket\_desc , client\_sock , c , read\_size;

**struct** sockaddr\_in server , client;

**char** client\_message[2000],data[10]={'\0'};

//Create socket

socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);

**if** (socket\_desc == -1)

{

printk("Could not create socket");

}

printk("Socket created");

//Prepare the sockaddr\_in structure

server.sin\_family = AF\_INET;

server.sin\_addr.s\_addr = INADDR\_ANY;

server.sin\_port = htons( 8888 );

//Bind

**if**( bind(socket\_desc,(**struct** sockaddr \*)&server , **sizeof**(server)) < 0)

{

//print the error message

printk("bind failed. Error");

**return** 1;

}

printk("bind done");

//Listen

listen(socket\_desc , 1);

//Accept and incoming connection

// printk("Waiting for incoming connections...");

c = **sizeof**(**struct** sockaddr\_in);

**while**(1)

{

printk("Waiting for incoming connections...");

//accept connection from an incoming client

client\_sock = accept(socket\_desc, (**struct** sockaddr \*)&client, (socklen\_t\*)&c);

**if** (client\_sock < 0)

{

printk("accept failed");

**return** 1;

}

printk("Connection accepted\n");

**while**(1)

{

**if**(token==0)

{

puts("token ==0 here , now reading data!");

read\_size = read(client\_sock , client\_message , 1);

**if**(strncmp(client\_message,str9,1)==0) //request all data

{

**if**(flag\_exit==1)

{

**continue**;

}

request=1;

token=1;

printf("receie request\n");

pthread\_mutex\_lock(&lock\_G\_VAI);

pthread\_cond\_wait(&cv\_G\_VAI,&lock\_G\_VAI);

pthread\_mutex\_unlock(&lock\_G\_VAI);

sprintf(msg\_back, "%d", G\_VAI);

write(client\_sock,msg\_back,4);

pthread\_mutex\_lock(&lock\_G\_AVI);

pthread\_cond\_wait(&cv\_G\_AVI,&lock\_G\_AVI);

pthread\_mutex\_unlock(&lock\_G\_AVI);

sprintf(msg\_back, "%d", G\_AVI);

write(client\_sock,msg\_back,3);

pthread\_mutex\_lock(&lock\_G\_heart);

pthread\_cond\_wait(&cv\_G\_heart,&lock\_G\_heart);

pthread\_mutex\_unlock(&lock\_G\_heart);

sprintf(msg\_back, "%d", G\_heart);

write(client\_sock,msg\_back,3);

}

**else** **if** (strncmp(client\_message,str3,1)==0) //request record

{

printf("request record\n");

pthread\_mutex\_lock(&lock\_trans);

strcpy(VAI\_trans[global\_counter+1],"8888");

strcpy(AVI\_trans[global\_counter+1],"888");

**for**(i=0;i<SIZE\_RECORD;i++)

{

write(client\_sock,VAI\_trans[i],4);

}

**for**(i=0;i<SIZE\_RECORD;i++)

{

write(client\_sock,AVI\_trans[i],3);

}

pthread\_mutex\_unlock(&lock\_trans);

}

**else** **if**(strncmp(client\_message,str0,1)==0) //stop

{

printf("receie stop\n");

doc\_init=1;

token=0;

}

**else** **if**(strncmp(client\_message,str1,1)==0) //begin

{

printf("receive begin\n");

flag\_start=1;

}

**else** **if**(strncmp(client\_message,str8,1)==0) //set AVI VAI

{

**char** data\_VAI[4];

printf("receive set\n");

**int** VAI\_Local,AVI\_Local;

read\_size = read(client\_sock , data\_VAI , 4);

sscanf(data\_VAI,"%d",&VAI\_Local);

printf("VAI\_Local = %d \n",VAI\_Local);

**if**(VAI\_Local > 9000)

VAI\_Local-=9000;

VAI\_s=VAI\_Local/1000;

VAI\_ms=VAI\_Local%1000;

read\_size = read(client\_sock , client\_message , 3);

**int** i;

**for**(i=0;i<3;i++)

data[i]=client\_message[i];

sscanf(data,"%d",&AVI\_Local);

**if**(AVI\_Local > 900)

AVI\_Local -= 900;

printf("AVI\_Local = %d \n",AVI\_Local);

AVI\_s=AVI\_Local/1000;

AVI\_ms=AVI\_Local%1000;

**if**(flag\_exit!=1)

doc\_init=1;

token=0;

}

**else** **if**(strncmp(client\_message,str2,1)==0) //disconnect

{

printf("reveive disconn\n");

**break**;

}

**else**

{

printf("It is not '9', receive data= %s \n",client\_message);

}

}

outtime.tv\_sec = 0;

outtime.tv\_nsec = 10\*1000000;

nanosleep(&outtime,NULL);

}

}

}

Explanation;

1. Once the communication thread is created, it would not be terminated.
2. The initialization of the socket is shown at the beginning of the thread. The IP of it is *192.168.0.67*, and the port is *8888*.
3. After initialization, it goes into a loop where the data is sent and received.

### Client:

Code:

% --- Executes on button press in Connect.

function pb\_conn\_Callback(hObject, eventdata, handles)

global t;

t = tcpip('192.168.0.67',8888);

fopen(t); %it create the socket and get connection

% --- Executes on button press in pb\_allRequest.

function pb\_allRequest\_Callback(hObject, eventdata, handles)

global t request\_AVI request\_all;

request\_AVI=0;

request\_all=1;

x=0;

y=0;

yy=0;

xcount=0;

axes(handles.axes1); %draw on the axes1

cla reset;

hold on;

xlim([0 7000])

ylim([-1.1 1.1])

xlabel('Time (ms)');

ylabel('Ventricular Pace Atrial Pace');

axes(handles.axes2); %draw on the axes2

cla reset;

hold on;

xlim([0 7000])

ylim([0 180])

xlabel('Time (ms)');

ylabel('Heart rate (/min)');

while( request\_all==1 )

req\_data='9'; %all data requested

fwrite(t,req\_data);

for i=1:3

if (i==1) %VAI data

data=fread(t,4)';

num=str2num(char(data));

if(num>9000)

num=num-9000;

end

else

data=fread(t,3)'; %AVI or heart rate data

num=str2num(char(data));

end

if(i==1) %plot atrial pace

x1=xcount+1: xcount+num+10; %the 'num' is the time of VAI

xcount=xcount+num+10; %10 is the error value used to show the peak

x=[x x1];

yy1=x1;

y1=linspace(0,0,num);

y2=linspace(1,1,10); %y-axis show the pace, y=1 means atrial pace

y=[y y1 y2];

axes(handles.axes1);

plot(x,y,'b');

elseif(i==2) % plot vertri pace

if(num>900)

num=num-900;

end

x1=xcount+1: xcount+num+10;

xcount=xcount+num+10;

if (x==0) % refresh the plot

yy1=x1;

else

yy1=[yy1 x1];

end

x=[x x1];

y1=linspace(0,0,num);

y2=linspace(-1,-1,10); %y=-1 means ventricular pace

y=[y y1 y2];

axes(handles.axes1);

plot(x,y,'b');

end

if (i==3) %plot heart rate

if (num<300)

heart\_fast=1;

else

heart\_fast=0;

heartRate=num/10;

end

yy1=linspace(heartRate,heartRate,numel(yy1));

yy=[yy yy1];

axes(handles.axes2);

plot(x,yy,'b');

end

if (xcount>7000) %reset figure

xcount=0;

x=0;

y=0;

yy=0;

axes(handles.axes1);

cla reset;

hold on;

xlim([0 7000]);

ylim([-1.1 1.1]);

xlabel('Time (ms)');

ylabel('Ventricular Pace Atrial Pace');

axes(handles.axes2);

cla reset;

hold on;

xlim([0 7000]);

ylim([0 180]);

xlabel('Time (ms)');

ylabel('Heart rate (/min)');

end

recev\_data(i)=num; %convert the num to string

if (num>999)

show(i,1:4)=num2str(recev\_data(i));

flag\_longVAI=1 ; %set the flag if the VAI is in 4 digits

elseif (i==1)

show(i,1:3)=num2str(recev\_data(i));

flag\_longVAI=0;

elseif(i==2 && num<100)

show(i,2:3)=num2str(recev\_data(i));

show(i,1)='0';

else

show(i,1:3)=num2str(recev\_data(i));

end

end

%show the data

if (flag\_longVAI==1)

set(handles.VAI\_text, 'String',show(1,1:4));

else

set(handles.VAI\_text, 'String',show(1,1:3));

end

set(handles.AVI\_text, 'String',show(2,1:3));

if(heart\_fast==1)

set(handles.HR\_text, 'String',show(3,1:3));

else

set(handles.HR\_text, 'String',show(3,1:2));

end

pause(0.1);

end

if (request\_all==0) %to stop the pacemaker

fwrite(t,'0');

pause(0.1);

end

% --- Executes on button press in Set bottom.

function pb\_set\_Callback(hObject, eventdata, handles)

global t;

req\_data='8'; %set VAI AVI value

fwrite(t,req\_data);

pause(0.1);

VAI\_value = get(handles.VAI\_text,'String'); %get the value from GUI

VAI\_value=str2num(VAI\_value);

if(VAI\_value<1000)

VAI\_value=VAI\_value+9000;

end

VAI\_value=num2str(VAI\_value);

AVI\_value = get(handles.AVI\_text,'String'); %get the value from GUI

AVI\_value=str2num(AVI\_value);

if(AVI\_value<100)

AVI\_value=AVI\_value+900;

end

AVI\_value=num2str(AVI\_value);

fwrite(t,VAI\_value);

pause(0.1);

fwrite(t,AVI\_value);

pause(0.1);

% --- Executes on button press in Disconnect.

function pb\_disconnect\_Callback(hObject, eventdata, handles)

global t;

fwrite(t,'2');

pause(0.5);

fclose(t);

% --- Executes on button press in Begin

function pb\_begin\_Callback(hObject, eventdata, handles)

global t;

fwrite(t,'1'); %begin

% --- Executes on button press in pb\_stop.

function pb\_stop\_Callback(hObject, eventdata, handles)

global flag\_stop request\_all;

flag\_stop=1;

request\_all = 0;

## 3.4 Debugging on Board

The pacemaker program should be running on the FPGA board after downloading the leon3 CPU to the board and the JTAG is used.

### Bitsteam download

Open the VIVADO and click Open Hardware Manager, and then click the Auto Connect. Right click the device and click Program Device, and download the leon3 bitstream file.

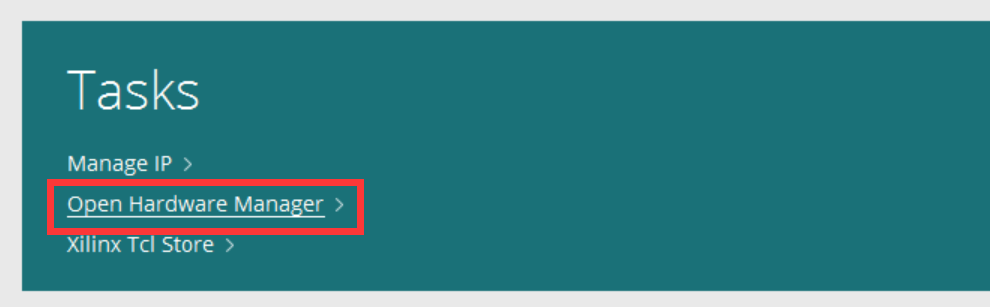


Fig. Open Hardware Manager

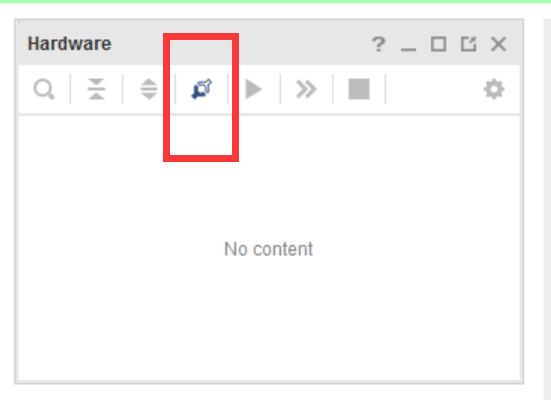


Fig. Auto Connect

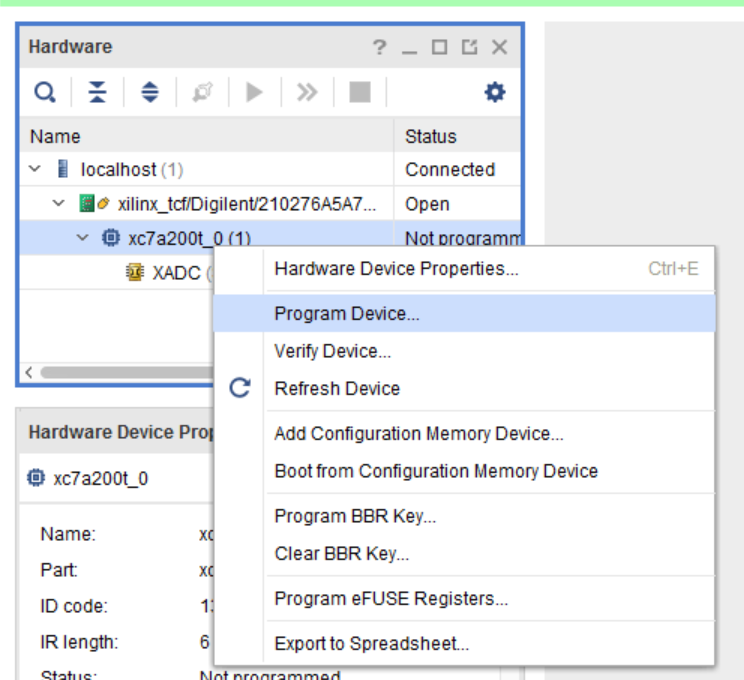


Fig. Program Device

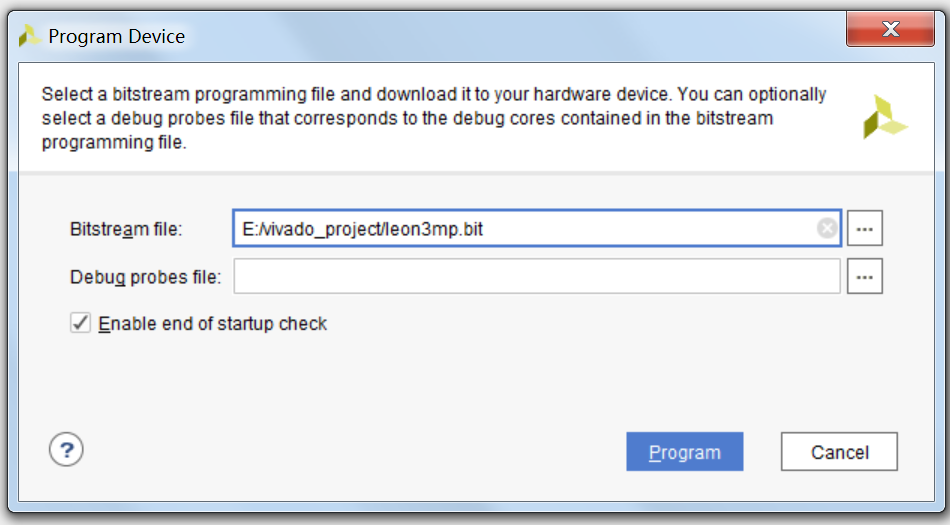


Fig. Download bitstream

### JTAG with GRMON

Grmon should be installed from <https://www.gaisler.com/index.php/downloads/debug-tools>. After the installation, using command line “–grmon –u –digilent” to connect the JTAG. The directory should be right. Use “-load filename” to load program into the board. After that use “run” to run the program.

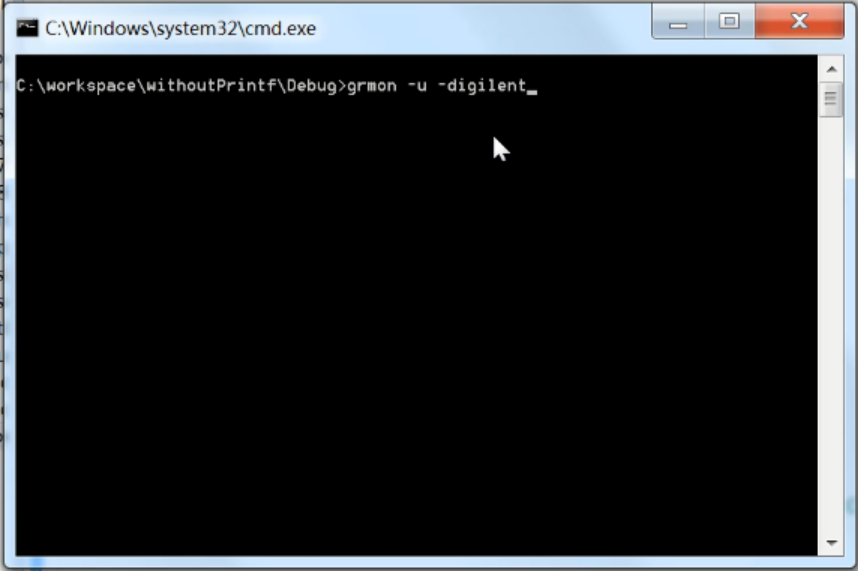


Fig. Start GRMON

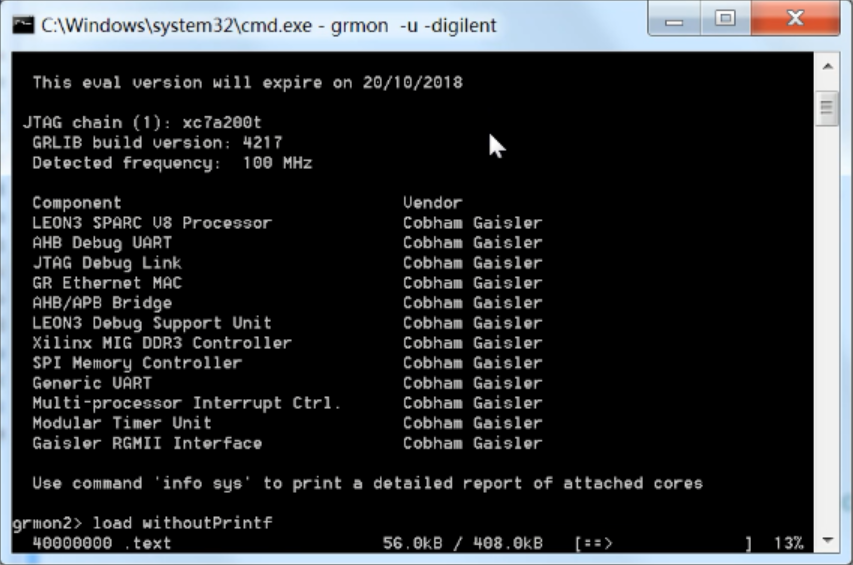


Fig. Load program

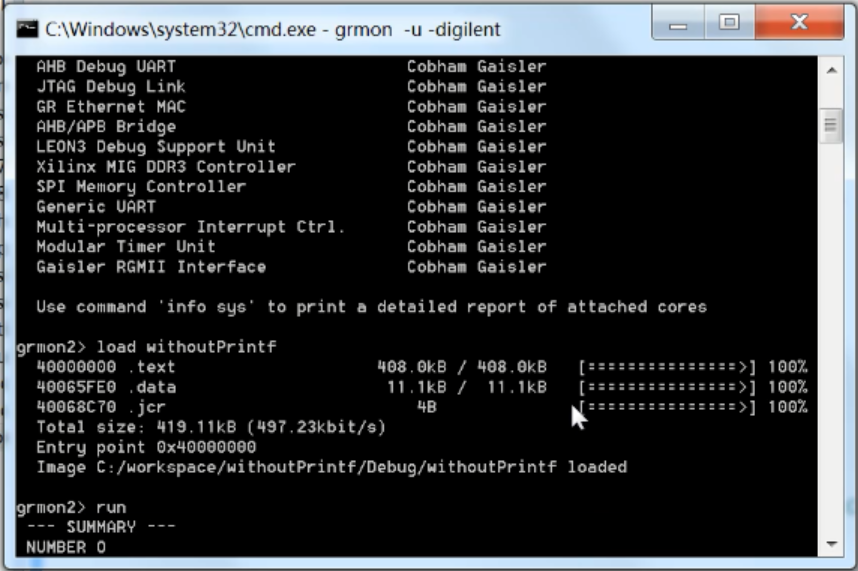


Fig. Run Program

# Chapter 4 MATLAB GUI

In this chapter, the GUI on the MATLAB would be introduced and specified. Users could set the VAI and AVI value via GUI. Real-time plots of the pace and heart rate are drawn and some data is shown.

## 4.1 Create a MATLAB GUI

A Matlab GUI could be created using the command line window (Fig. 9), entering “guide”, and then chose the “Blank GUI (Default)” (Fig. 10). Using the tools on the left side of Fig. 11 improve the function of GUI.

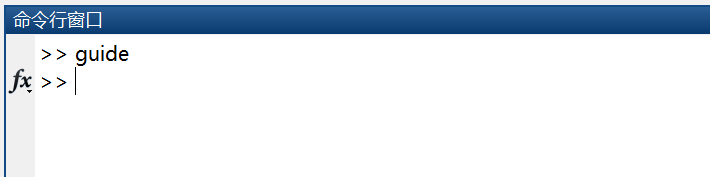


Fig. MATLAB command line window

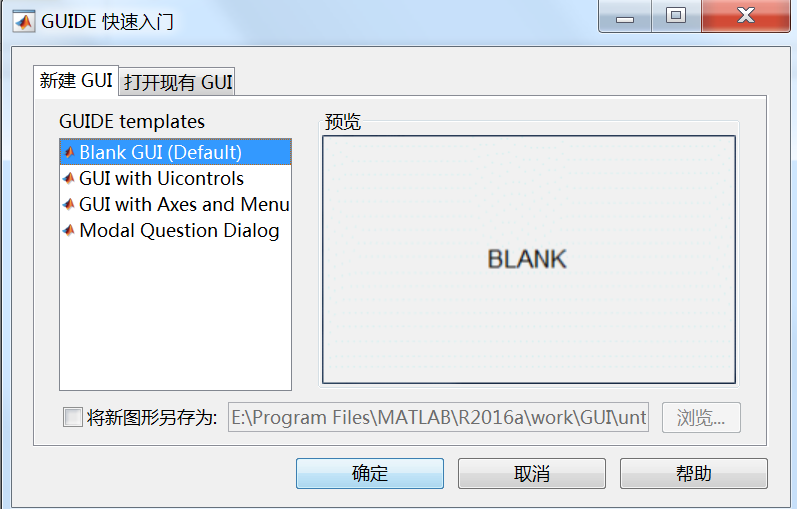


Fig. Create a blank GUI

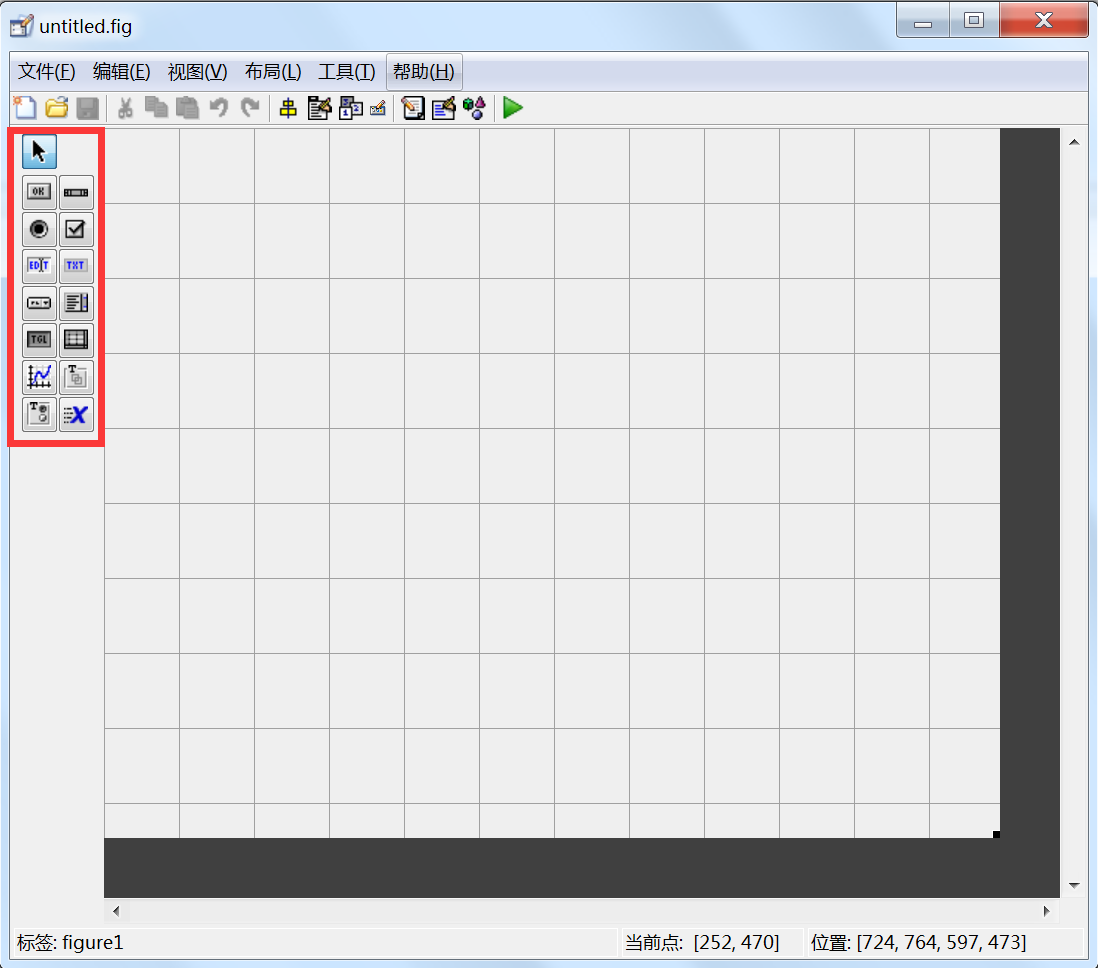


Fig. Edit the GUI

## 4.2 GUI of the Pacemaker

In this application, a GUI is created, using six bottoms to send different command, two axes to show the real-time plots, three textboxes to show the data and two of them could be used to set AVI and VAI time. The GUI is shown in Fig. 12.

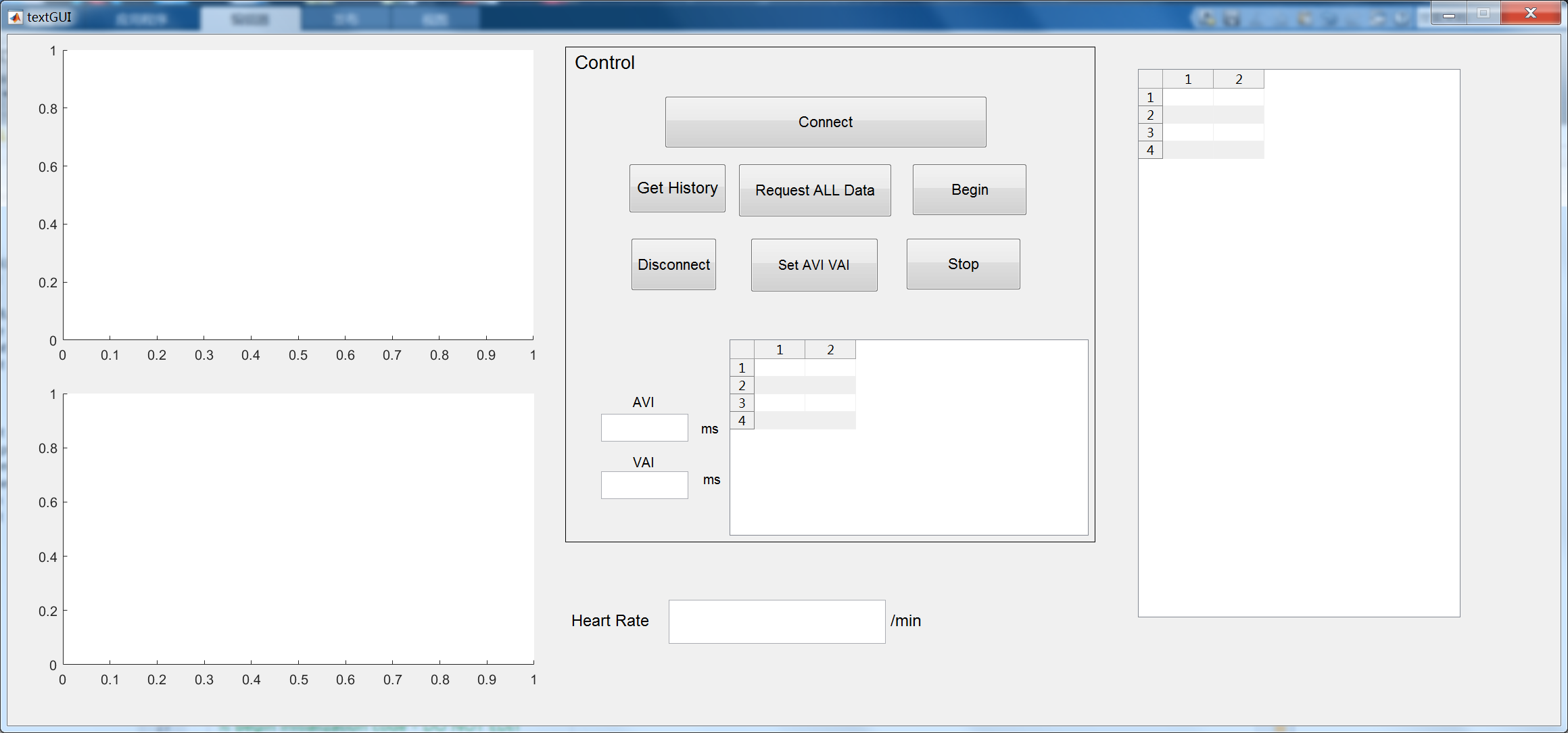


Fig. GUI of the pacemaker

### Instruction

1. Start the GUI and pacemaker
2. Click the bottom ‘Connect’ when the initialization is established in pacemaker.
3. Click the bottom ‘Get History’ to get the record data of AVI and VAI of the patient up to the number of 1024.
4. Click the bottom ‘Request ALL Data’ to request VAI, AVI and heart rate data from pacemaker, after which the axes1 and axes2 would begin to draw the heart pace and heart rate while these three data would be shown in the textboxes and recorded in the list box.
5. To stop the pacemaker, click the bottom ‘Stop’.
6. After the pacemaker stops, AVI and VAI could be set. To set them click the ‘Set AVI VAI’ bottom.
7. Click the bottom ‘Begin’ to start the pacemaker with the new values of AVI and VAI set.
8. Click the bottom ‘Disconnect’ to disconnect with the pacemaker.