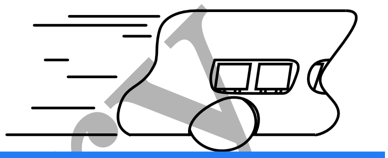
Khepera^2



Programming manual

Version 0.2
June 2002

Documentation Author

Pierre Bureau for K-Team S.A.
Ch. de Vuasset, CP 111
1028 Préverenges
Switzerland

email: info@k-team.com
Url: www.k-team.com

TABLE OF CONTENTS

1 INTRODUCTION

This document is a preliminary beta version of the Khepera Programming Manual. It is provided "as is" without warranty of any kind. The content is subject to change without notice and very likely to contain errors, misspelling problems and formatting bugs.

Error reports, suggestion for improvements, comments, and any other form of feedback are welcome. This document is released under the "GNU Free Documentation License" and any Khepera user can contribute to, modify, and distribute this manual at will.

This document's LaTeXsources are available for download from www.k-team.com, and are released under the GFDL as well.

All feedback can be send to info@k-team.com.

1.1 GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1.2 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

1.3 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute.

However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

1.4 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

1.5 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the

Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- In any section entitled “Acknowledgements” or “Dedications”, preserve the section’s title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.
- Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

1.6 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

1.7 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

1.8 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

1.9 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

1.10 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

1.11 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ”or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

2 PROGRAMMING BASICS

The Khepera BIOS system includes a complete Application Programming Interface for C, and assembly language, development. Using this API, programmers can easily develop Khepera applications without an extensive knowledge of the robot's hardware.

Low level functions are provided for interactions with motors and sensors, for communications with extensions or a host computer, and for basic multitasking management.

As applications are not usually focused on motor control, classic PID Controllers for motors are implemented in the BIOS, either for speed or position control. However, functions are also provided for applications to use their own controller implementation if required. Simple methods for interactions with proximity sensors and motors encoder are provided for applications to collect data.

Any elaborated application will need to communicate with the outside world. Basically, the Khepera can communicate with a computer using a standard serial link. When using Khepera extensions, the serial link can be diverted to a wireless communication or another channel. Applications can also communicate with extensions, such as new sensors or a gripper, using a simple set of functions and protocol.

Multitasking management includes functions to install, kill, and suspend tasks. Such an environment is helpful for complex application development that should perform several actions in parallel. Basic communication between tasks is also supported, as well as event management.

2.1 Multitask Environment

An application using a multitask environment is quite different from a standard stand-alone application. Several tasks can be executed in parallel, eventually communicating together and using shared resources.

Applications must first deal with task installation, they can be designed to run as monolithic applications in a single task, but complex applications are usually much easier to develop using a multitask approach.

As soon as two, or more, tasks are used, communication may be necessary. Tasks might need to exchange data collected in a task but required in another. Events can be signaled to other tasks for synchronization or simple information purpose.

Because they are running on the same system, tasks may use the same Hardware and Software resources. To avoid potential conflicts, when two tasks are trying to use the same resource, protection mechanisms are required.

2.1.1 Task General Organization

The TIM micro-kernel is in charge of tasks organization. Depending on each task status, it is placed in one of the five different task list. The tasks status can be modified by calling given functions or when given events occurs. These mechanisms are detailed in the following sections.

Empty list: The empty list can be considered as a container for free task descriptors. Every time a new task is installed, a descriptor is "removed" from the empty list and placed in the execution list. The empty list contains a maximum of thirty two task descriptors.

Execution list: The execution list contains all the active tasks. The first task descriptor (or task zero) is always present in the execution list and its status cannot be changed. Every 5ms the scheduler interrupt switches from the current task to the next available one in the execution list.

Wait list: Any task which has been suspended for a given length of time is placed in the wait list. These tasks cannot be scheduled except when the given timeout is past.

Event list: When task are suspended until a event occurs, they are placed in the event list. Events are generated from other tasks, and are usually used as a synchronization method.

Sync list: When a task is suspended until an external event occurs, it is placed in the sync list. These tasks are not scheduled until one of the given events occur. External events are generated by peripherals.

2.1.2 Task Management

Launching Task

An application should first launch all its tasks before switching to multitask management. This is usually done within the main function, altogether with all application dependent initialization.

The `main()` is the application entry point, as in any C application, however the code within the main function is not considered as a task. On the other hand, the scheduler can interrupt the main code and switch to another task, as soon as any new task is installed. Because of this situation, the main function is not really suitable for application code execution, a typical main function should only:

- Execute initialization code
- Eventually suspend the scheduler
- Launch all the application tasks
- Eventually wake up the scheduler
- Terminate

A typical main function code is such as:

```
#define STACK_SIZE 800
static int32 vIDProcess[N+1];

/** Main entry point **/
int main(void) {
  int32 status;
  static char prName_0[] = "User Process 0";
  [...]
  static char prName_N[] = "User Process N";

  /** Initialization **/
  all_init();

  /** Suspend scheduler if needed **/
  tim_lock();

  /** Install Task 0 **/
  status = install_task (prName_0, STACK_SIZE, process_0);
  if (status == -1)
  {
     printf("Error launching process 0\n\r");
     exit (0);
  }
  vIDProcess[0] = (uint32) status;
  [...]
  status = install_task (prName_N, STACK_SIZE, process_N);
  if (status == -1)
  {
```

```
     printf("Error launching process N\n\r");
     exit (0);
  }
  vIDProcess[N] = (uint32) status;

  /** Wake up scheduler if needed **/
  tim_unlock();

  return 0;
}
```

Tasks which are not necessary at start-up should also be launched. Such tasks will first be suspended, waiting for an event to wake them up (refer to section 2.1.2 for further details).

New tasks can also be launched at any time, from another task or from the main if not terminated. For most applications, this is however not recommended.

When the application cannot tell which tasks are necessary from start-up, and if the system is short on memory, then it might be necessary to wait until a task is required before installing it.

As code to install a new task is always the same, to simplify next examples, the following macro will be used. *This macro is not syntactically correct regarding C programming*, and is only a writing shortcut.

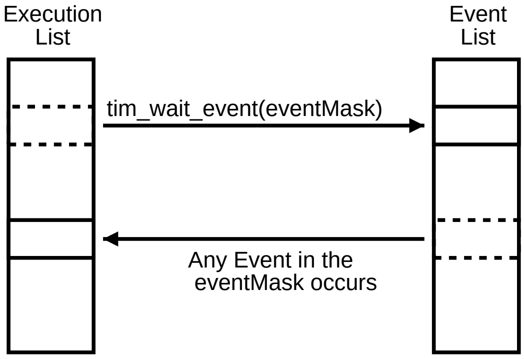
```
  INSTALL_NEW_TASK(N)
  {
    status = install_task ("Process name N", STACK_SIZE, process_N);
    if (status == -1)
    {
       printf("Error launching process N\n\r");
       exit (0);
    }
    vIDProcess[N] = (uint32) status;
  }
```

Suspending Task

Whenever a task is not needed for a while, it should be suspended rather than killed.

The `tim_suspend_task()` function is useful to suspend a task for a given amount of time, it is actually preferred for periodic tasks (see section 2.1.2). When a task should be suspended for an unknown delay, until an event occurs, the `tim_wait_event()` function should be used.

After calling `tim_wait_event()`, task execution is suspended, and the task is placed in the Event list. The scheduler then switch to another task in the execution list. Until one of the awaited event occurs, the suspended task is never scheduled, then it is placed back in the Execution list and can be executed again.



Events are generated using `tim_generate_event()`, any task can call this function at any time to generate an event. The identifier of an event is always the same as the identifier of the task from which it has been generated. A task cannot generate different events, a more elaborated event communication system must be implemented if required (see section 2.1.5).

When an event is generated, any task in the Event list waiting for this event are placed back in the execution list. The scheduler is then in charge of task switching as usual.

The following code is a simple example where one task is waiting for a request from another before executing. This is the usual mechanism for task synchronization. This program is a simple demonstration and requires a working serial link connection.

```
#define STACK_SIZE 800
static int32 vIDProcess[3];

/** Code for Task 0 **/
static void process_0(void) {
  unsigned count = 0;
  /** Run continuously **/
  for(;;) {
    /** wait for synchro **/
    tim_wait_event(vIDProcess[1]|vIDProcess[2]);
    printf("Event %u received.\r\n",count);
    count++;
  };
  return 0;
}
```

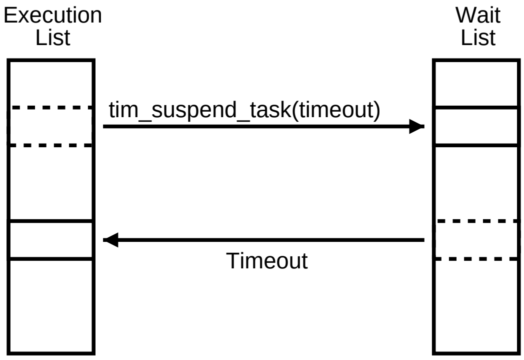
```
/** Code for Task 1 **/
static void process_1(void) {
  /** Run continuously **/
  for(;;) {
    /** Generate random events **/
    rand_num = 1+(int)(1000.0*rand()/(RAND_MAX+1));
    if(rand_num == 999)
      tim_generate_event();
  }
  return 0;
}

/** Code for Task 2 **/
static void process_2(void) {
  /** Run continuously **/
  for(;;) {
    /** Generate event when 'a' is pressed **/
    if( 'a' == fgetc(stdin))
      tim_generate_event();
  }
  return 0;
}

/** Main entry point **/
int main(void) {
  int32 status;

  tim_reset();
  INSTALL_NEW_TASK(0);
  INSTALL_NEW_TASK(1);
  INSTALL_NEW_TASK(2);
  return 0;
}
```

Periodic Task

A periodic task execution is based on a given timeout rather than on the normal scheduler interrupt mechanism. The TIM micro kernel provides the `tim_suspend_task()` function to suspend task execution for a given length of time.

After calling `tim_suspend_task()`, a task execution is suspended, and its descriptor is placed in the Wait list. A task in the Wait list will be wake up by the Real Time Clock Interrupt that occurs every millisecond, when the given timeout is past. Then it is placed back in the Execution list until suspended again.

A suspended task can be useful for Real Time processing, or for a simple execution pause. It is easily implemented, as in the following example, where the `periodic_action()` function is executed every 500ms:



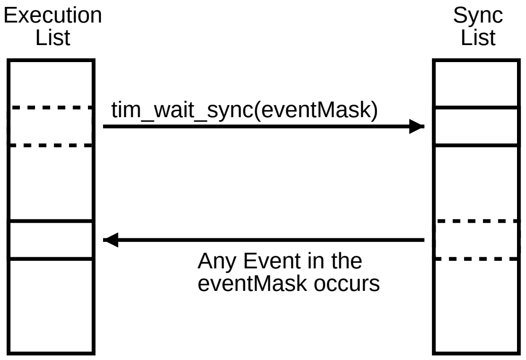
```
#define STACK_SIZE 800
static int32 vIDProcess[1];

/** Code for Task 0 **/
static void process_0(void) {
  /** Run continuously **/
  for(;;) {
    /** Suspend task for 500ms **/
    tim_suspend_task(500);
    periodic_action();
  };
  return 0;
}

/** Main entry point **/
int main(void) {
  int32 status;

  tim_reset();
  INSTALL_NEW_TASK(0);
  return 0;
}
```

External Synchronization

Task execution can also be suspended until an external event occurs. External events can be messages from the serial link, sensors synchronization, motor controller interrupts and more. The `tim_wait_sync()` function is used to suspend a task execution and to specify awaited events.

After calling `tim_wait_sync()` a task is placed in the Sync List. It is not scheduled until one of the given event occurs, then it is placed back in the execution list.

The list of awaited events is given as a sum of sync mask, this function call in C should look such as `tim_wait_sync(2^1|2^5|2^3)`. Available events are described in the following table:



SyncMask	**Event**	**Description**
2^0	PID sample	A new motor command is calculated
2^1	On target	Target position is reached
2^2	MSG sent	Message sent from the MSG manager
2^3	MSG receive	Message received by the MSG manager
2^4	SER sent	Message sent from the SER manager
2^5	SER receive	Message received by the SER manager
2^6	IR Sensor Sync	One of the IR sensors has been sampled
2^7	IR0 Sensor Sync	IR Sensor 0 has been sampled
2^8	IRQ interrupt	Interrupt received on the IRQ line of the K-Bus

Applications using hardware resources, can use this mechanism to synchronize with these resources. The following example displays a message on every IR sensor 0 Sync, indicating that the eight sensors have been scanned.

```
#define STACK_SIZE 800
static int32 vIDProcess[1];

/** Code for Task 0 **/
static void process_0(void) {
  uint32 time;
  /** Run continuously **/
  for(;;) {
    tim_wait_sync(2^7);
    time=tim_get_ticcount();
    printf("Sensor scan: %lu\n\r",time);
  };
  return 0;
}

/** Main entry point **/
int main(void) {
  int32 status;

  tim_reset();
```

```
  INSTALL_NEW_TASK(0);
  return 0;
}
```

Killing Task

Some applications may need a task for a limited period only, in that case the task should be terminated. A task can terminate itself simply by returning as a common C function or by using the `exit()` function. The `kill_task()` function can also be called from a task to terminate another, depending on the application structure.

Whenever a task is not needed for given period only, it should be suspended rather than killed, please refer to the above sections for further information. However, it can be necessary to kill a task for memory saving purposes, even if the task is likely to be installed again latter.

The following example of a task terminating itself after a given period of time and killing a second task at the same time.

```
#define STACK_SIZE 800
static int32 vIDProcess[2];

/** Code for Task 0 **/
static void process_0(void) {
  uint32 time;
  /** Run continuously **/
  for(;;) {
    tim_suspend_task(500);
    time++;
    var_change_led(1);
    if(time > 20)
    {
      kill_task(vIDProcess[1]);
      exit(0);
    }
  };
  return 0;
}

/** Code for Task 1 **/
static void process_1(void) {
  /** Run continuously **/
  for(;;) {
    tim_suspend_task(500);
    var_change_led(0);
  };
  return 0;
}
```

```
/** Main entry point **/
int main(void) {
  int32 status;

  tim_reset();
  INSTALL_NEW_TASK(0);
  INSTALL_NEW_TASK(1);
  return 0;
}
```

2.1.3 Process Information

Programmers can get more information about running process using `tim_get_task_des_ptr()`. This function returns a pointer to the first `procDesc` structure in a linked list of all process descriptors.

The `procDesc` structure is described in the following code. Modifying the data in a descriptor will generally cause system faillures, applications should only read these information.

```
/** Process State Masks **/
#define         BPRES           0  /* process active */
#define         BLOCK           1  /* process locked */

/** Process Descriptor Structure **/
struct procDesc
{
  uint32          oIDProc;      /* process ID    */
  uint8           oStaProc;     /* process state */
  uint8           oReserve;     /* reserve       */
  uint32          oMskEvent;    /* event mask    */
  uint32          oMskSync;     /* synchro mask  */
  PROCDESC        *oPtrCurrent; /* ptr on the current process    */
  PROCDESC        *oPtrBack;    /* ptr on the previous process   */
  PROCDESC        *oPtrForward; /* ptr on the next process       */
  char            *oPtrTextID;  /* ptr on process name           */
  uint32          oTStopProc;   /* wait time for suspended proc  */
  uint32          *oPtrStkPro;  /* process stack                 */
};

/** Example loop to print all active process name **/
uint32 i;
PROCDESC * ptrProc;

for(i=0; i<32; i++)
{
  ptrProc = tim_get_task_des_ptr()
  ptrProc = ptrProc + i;
  if(ptrProc->oStaProc == (1 << BPRES))
    printf("%s",ptrProc->oPtrTextID);
}
```

2.1.4 Critical Resources

Any resource that can be used by two or more tasks is considered as critical. The system ressources can be software ressources, such as variable or files, and hardware ressources, such as peripherals or communication channels.

Access to critical ressources must be protected to avoid potential conflicts between tasks. To illustrate such a conflict, if `int A` is a shared variable between several tasks, and if one of these tasks executes the following operation: `C=(A+B)*A`. If the addition `A+B` is executed, then the task is interrupted and a new value is written to `A` by another task. When the scheduler swithches back, the opperation `C=(A+B)*A` is completed, but the result is invalid as `A` has been modified during the process.

Scheduler Suspend

The easiest way to protect shared resources is to lock the scheduler as long as the resource is required. A task locking the scheduler will be executed until unlocking it and other tasks do not get the opportunity to mess with shared resources or data.

On the other hand, periodic tasks, external event synchronization and all the others are ignored when the scheduler is locked. That is why applications should usually avoid to lock the scheduler for a long period of time, and moreover the scheduler should never gets locked for an undetermined period of time.

With the next example, a set of variable is shared between the tasks. `tim_lock()` and `tim_unlock()` are called to protect any access to shared data, which are updated using the user defined function `data_update()`.

```
#define STACK_SIZE 800
static int32 vIDProcess[2];

/** Shared data **/
int32 data0,data1;

/** Code for Task 0 **/
static void process_0(void) {
  int32 d0,d1;

  for(;;) {
    tim_lock();
    data_update();
    d0=data0;
    d1=data1;
    tim_unlock();
```

```
    [... Process data ...]
  }
}

/** Code for Task 1 **/
static void process_1(void) {
  int32 d0,d1;

  for(;;) {
    tim_lock();
    d0=data0;
    d1=data1;
    tim_unlock();

    [... Process data ...]

    tim_lock();
    data0=d0;
    data1=d1;
    tim_unlock();
  }
}

/** Main entry point **/
int main(void) {
  int32 status;

  tim_reset();
  INSTALL_NEW_TASK(0);
  INSTALL_NEW_TASK(1);
  return 0;
}
```

Semaphore Implementation

To avoid locking the scheduler during long periods of time, semaphores can be implemented for critical resources protection. A semaphore can be seen as a token corresponding to a single resource.

Each time a resource is needed, the task must first reserve the corresponding token. When finished with the resource, the task must free the token. If another task needs the same resource while the token is reserved, it must wait until the token is available.

The following code is a very basic example of semaphore implementation, with a maximum of 32 different tokens. Much more complex implementations are available, using queue for requesting tasks, priorities, and multiple tokens per semaphore.

```
#define STACK_SIZE 800
static int32 vIDProcess[2];
```

```
uint32 semaphoreList= 0xFFFFFFFF;

/** Semaphore initialization **/
void sem_init(unsigned sem_id)
{
  if( sem_id > 31)
    return BAD_ID;
  tim_lock();
  semaphoreList&= (!(0x00000001 << sem_id));
  tim_unlock();
}

/** Semaphore Request **/
int sem_p(unsigned sem_id)
{
  tim_lock();
  if( (0x00000001 << sem_id) & semaphoreList)
  {
    /** Token available - reserve token **/
    semaphoreList&= (!(0x00000001 << sem_id));
    tim_unlock();
    return 1;
  } else {
    /** Token reserved - do nothing **/
    tim_unlock();
    return 0;
  }
}

/** Semaphore Release **/
void sem_v(unsigned sem_id)
{
  tim_lock();
  semaphoreList&= (0x00000001 << sem_id);
  tim_unlock();
}

/** Code for Task 0 **/
static void process_0(void) {
  for(;;)
  {
    [... Task activities ...]

    /** Wait until resource 0 is available **/
    while( !sem_p(0) );
    use_resource();
    sem_v(0);

    [... Task activities ...]
  }
}

/** Code for Task 1 **/
```

```
static void process_1(void) {
  for(;;)
  {
    [... Task activities ...]

    /** Try to access resource 0 and continue **/
    if( sem_p(0) )
    {
      use_resource();
      sem_v(0);
    }

    [... Task activities ...]
  }
}

int main(void) {
  int32 status;

  tim_reset();

  sem_init(0);

  INSTALL_NEW_TASK(0);
  INSTALL_NEW_TASK(1);
  return 0;
}
```

2.1.5 Task Communication

Event generation, problem: no acknowledge, if the task is not waiting the even might be lost.

Non Blocking FIFO communication

mailbox

Differentiated Event Communication

2.2 Motor Control

The Khepera robot is equipped with two DC motors and incremental encoders. Controlling the robot's movements is a matter of controlling the motors, using the encoders feedback.

An easy and efficient way to control DC motors is to use a Pulse Width Modulation signal. When applying such a signal to a DC motors, the speed is roughly proportional to the PWM ratio. However, because of variations between motors, temperature influence, and non constant load, this is not enough to control the robot's movement.

That is why a controller using the encoders feedback is required. The Khepera's API provides two controllers, one for position control and the other for speed control. Both controllers are classical PID implementations.

Further details are given in the Khepera User Manual and extensive documentation on motor PWM control and PID equations is available from various sources.

2.2.1 Using Speed Control

Figure 2.1 shows an overview of the speed controller implementation. Applications can setup this controller using the following BIOS functions:

`-mot_config_speed_1m()`	To set the PID coefficient
`-mot_new_speed_1m()`	To set speed command for one motor
`-mot_new_speed_2m()`	To set speed commands for both motors
`-mot_get_speed()`	To get motors speed

The speed controller is a very classic PID loop, the speed command is compared with the calculated speed to get the error. Then a PID equation is used to calculate a new PWM command and the new calculated speed is obtained by position feedback derivation.

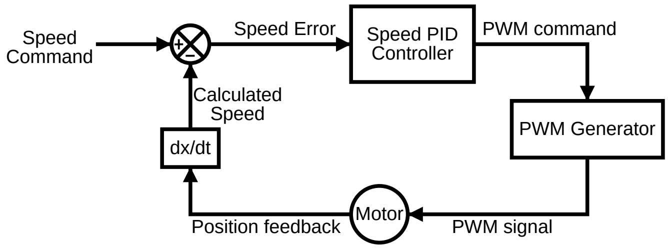


Figure 2.1: Speed Control Overview

Setting Motors Speed

Applications can set the speed command and configure the PID coefficient using provided BIOS functions. A new PWM command is calculated every 10ms, using the given speed command, and PID coefficient.

The speed command, not the speed itself, is modified, so that depending on PID tuning, speed change might be very slow. Unpredictable results may even occur, if the PID configuration is not stable.

```
#define STACK_SIZE 800
#define SPEED 20
static int32 vIDProcess[1];

/** Code for Task 0 **/
static void process_0(void) {

  /** Set PID coefficient for motor 0 **/
  mot_config_speed_1m(0,Kp,Ki,Kd);

  /** Set PID coefficient for motor 1 **/
  mot_config_speed_1m(1,Kp,Ki,Kd);

  /** Set speed for motor 0 **/
  mot_new_speed_1m(0,SPEED);

  /** Set speed for motor 1 **/
  mot_new_speed_1m(1,SPEED);

  /** Run continuously **/
  for(;;) {};

  return 0;
}

/** Main entry point **/
int main(void) {
  int32 status;
  static char prName_0[] = "User Process 0";

  /** Reset MOT module **/
  mot_reset();

  /** Install Task 0 **/
  INSTALL_NEW_TASK(0);
  return 0;
}
```

This very simple example just sets the speed command for both motors to a constant value a then runs forever. Both motors will be driven to the given speed until the Khepera power is switched off or until the task is killed using a serial link command.

In that simple case, it is actually easier and safer to set both motors speed at the same time, by replacing code for task 0 with:

```
/** Code for Task 0 **/
static void process_0(void) {

  /** Set speed for both motors **/
  mot_new_speed_2m(SPEED,SPEED);

  /** Run continuously **/
  for(;;) {};
}
```

Speed Control Loop Example

For all but the most simple applications, simply setting the speed is not enough. The following example dynamically sets the speed command every SUSPEND milliseconds, according to the ambient light. The brighter is the ambient light, the faster goes the Khepera, until it stops moving in complete darkness. Please refer to section 2.3 for details concerning `Get_Max_Ambient()` function.

The second task is simply displaying the current speed every 500ms for monitoring purpose. If the Khepera is not connected to a terminal, with a proper serial link configuration, this task will have no effect.

Much more complex speed control algorithm might be necessary but the speed command update principle will stay the same.

```
#define STACK_SIZE 800
#define SUSPEND 200
3#define MAX_AMBIENT 240
static int32 vIDProcess[2];

/** Code for Task 0 **/
static void process_0(void) {
  uint32 speed;
  uint32 ambient;

  /** Set PID coefficient for motor 0 **/
  mot_config_speed_1m(0,Kp,Ki,Kd);

  /** Set PID coefficient for motor 1 **/
  mot_config_speed_1m(1,Kp,Ki,Kd);

  /** Run continuously **/
  for(;;) {
    /** Suspend the Task for SUSPEND ms **/
```

```
    tim_suspend_task(SUSPEND);

    /** Get the maximum Ambient Light **/
    ambient=Get_Max_Ambient();

    /** Calculate new speed **/
    speed=abs(MAX_AMBIENT-ambient);

    /** Set speed for both motors **/
    mot_new_speed_2m(speed,speed);
  };
}

/** Code for Task 1 **/
static void process_1(void) {
  uint32 speed1,speed0;

  /** Run continuously **/
  for(;;) {
    /** Suspend the Task for 500ms **/
    tim_suspend_task(500);

    /** Get motors speed **/
    speed0=mot_get_speed(0);
    speed1=mot_get_speed(1);

    /** Print speeds to serial line **/
    printf("Speed m0:%lu m1:%lu\n\r",speed0,speed1);
  }
}

/** Main entry point **/
int main(void) {
  int32 status;
  static char prName_0[] = "User Process 0";

  /** Reset MOT module **/
  mot_reset();

  /** Install Tasks **/
  INSTALL_NEW_TASK(0);
  INSTALL_NEW_TASK(1);
  return 0;
}
```

Setting the motors speed is pretty easy, but the result of a `mot_new_speed_xx()` command finally depends on the PID settings and on general PID dynamics. Nevertheless, PID default setting is good enough for most applications and proven to be stable.

2.2.2 Using Position Control

Figure 2.2 shows an overview of the position controller implementation. Applications can setup this controller using the following BIOS functions:

`-mot_config_position_1m()`	To set PID coefficient
`-mot_config_profil_1m()`	To set the speed profile
`-mot_get_position()`	To get motors position
`-mot_put_sensor_1m()`	To calibrate one sensor's position
`-mot_put_sensor_2m()`	To calibrate both sensors position
`-mot_new_position_1m()`	To set target position for one motor
`-mot_new_position_2m()`	To set target position for both motors

The position controller is a bit more complicated than the speed controller. It is using a speed profile to generate a position command according to the given target position and position feedback. The target position, set with `mot_new_position_xx()`, is not directly used as a command for the PID controller.

The PID loop itself is very similar to the speed control loop. A new PWM command is calculated every 10ms according to the position error and the PID coefficient.

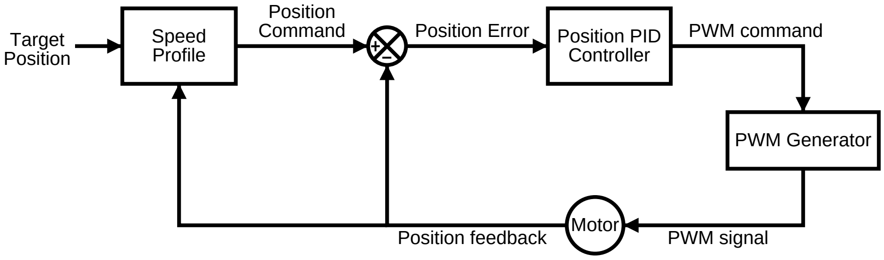


Figure 2.2: Position Control Overview

The Speed Profile

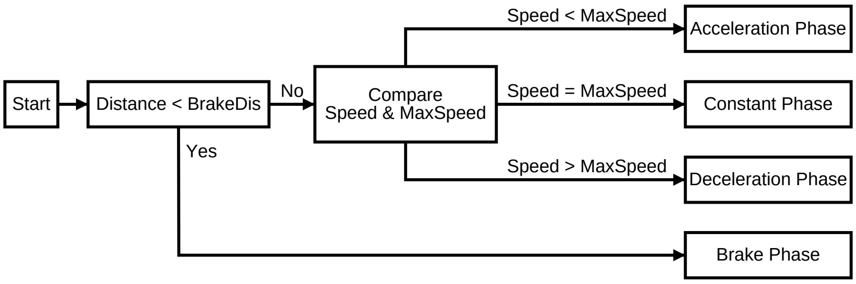
The speed profile is used to generate the position command. Every time a new PWM command is calculated, the distance from target is first estimated.

$$Distance = TargetPos - CurrentPos$$

As well as the braking safety distance.

$$BrakeDis = \frac{Speed^2}{2 * MaxAccel}$$

Then, according to the profile configuration and the current speed, the following algorithm determines which phase is to be applied. Each phase implies a different position command calculation.



Acceleration Phase: Increase current speed

$$\begin{aligned} Speed =& \quad Speed + Acceleration \\ PositionCommand =& \quad CurrentPos + Speed \end{aligned}$$

Constant Phase: Keep current speed

$$\begin{aligned} Speed =& \quad Speed \\ PositionCommand =& \quad CurrentPos + Speed \end{aligned}$$

Deceleration Phase: Decrease current speed

$$\begin{aligned} Speed =& \quad Speed - Acceleration \\ PositionCommand =& \quad CurrentPos + Speed \end{aligned}$$

Brake Phase: Override max. acceleration and brake

$$\begin{aligned} Accel =& \quad \left|\frac{Speed^2}{2 * BrakeDis}\right| \\ Speed =& \quad Speed - Accel \\ PositionCommand =& \quad CurrentPos - Speed \end{aligned}$$

For every sampling period, Khepera's next position is calculated. This method is finally generating a trajectory, so that the Khepera is position controlled rather than speed controlled.

The profile parameters are set using:
`mot_config_profil_1m(motorNb, maxSpeed, maxAccel)`
where `maxSpeed` and `maxAccel` are the values to be used in the above algorithm, for the given motor.

Setting a Target Position

Applications can set a new target position, and configure the PID controller using the provided BIOS functions. The position command itself is generated with the speed profile, according to the profile settings.

```
#define STACK_SIZE 800
#define DIST 100
static int32 vIDProcess[1];

/** Code for Task 0 **/
static void process_0(void) {
  int32 pos0,pos1;

  /** Set PID coefficient for motor 0 **/
  mot_config_position_1m(0,Kp,Ki,Kd);

  /** Set PID coefficient for motor 1 **/
  mot_config_position_1m(1,Kp,Ki,Kd);

  /** Set Profile for motor 0 **/
  mot_config_profil_1m(0,maxSpeed,maxAccel);

  /** Set Profile for motor 1 **/
  mot_config_profil_1m(1,maxSpeed,maxAccel);

  /** Get motors position **/
  pos0 = mot_get_position(0);
  pos1 = mot_get_position(1);

  /** Set target position for motor 0 **/
  mot_new_position_1m(0,pos0+DIST);

  /** Set target position for motor 0 **/
  mot_new_position_1m(1,pos1+DIST);

  /** Run continuously **/
  for(;;) {};
}

/** Main entry point **/
int main(void) {
  int32 status;
  static char prName_0[] = "User Process 0";

  /** Reset MOT module **/
  mot_reset();

  /** Install Task 0 **/
  INSTALL_NEW_TASK(0);
  return 0;
}
```

This simple example just sets a target position for each motors, and then waits forever. Once target position is reached, the Khepera should not move anymore.

As for speed control, `mot_new_position_2m()` can be used to set target position for both motors at the same time. The target position should always be set relative to the current position, as the starting position is unknown (except if using `mot_put_sensors_xx()`, see below).

Position Control Loop Example

When setting a new target position, the previous target is canceled. There is no "target position stacking", so that for most applications a position control loop is necessary.

Before setting a new goal, an application should check for the controller status, to make sure target position is reached. Depending on the speed profile configuration, and target distance, the target may be approached very slowly, but this method is preferred when final positioning is critical.

`mot_put_sensors_xx()` can be used to avoid referencing target positions relative to the current measured position. This will force application to keep track of the robot's absolute position, but can simplify code in certain cases. Using `mot_put_sensors_xx()` is completely optional.

The following example is a possible implementation for a position control loop.

```
#define STACK_SIZE 800
static int32 vIDProcess[2];
/** Shared data between different process **/
int32 abs_pos0,abs_pos1;
int32 tgt_pos0,tgt_pos1;
int8  isEmpty;

/** Code for Task 0 **/
static void process_0(void) {
  int32 tgt_pos0,tgt_pos1;
  int32 meas_pos0,meas_pos1;
  int32 status0,status1;
  /** mask for testing bit 18 of motors status **/
  int32 bitmask=0x00040000

  /** Set PID coefficient for both motors **/
  mot_config_position_1m(0,Kp,Ki,Kd);
  mot_config_position_1m(1,Kp,Ki,Kd);

  /** Set Profile for both motors  **/
  mot_config_profil_1m(0,maxSpeed,maxAccel);
  mot_config_profil_1m(1,maxSpeed,maxAccel);
```

```
  /** Run continuously **/
  for(;;) {
    /** Check if a new position is available **/
    if(isEmpty)
            tim_wait_event(vIDProcess[1]);

    /** Protect access to shared data and Get new target position **/
    tim_lock();
    tgt_pos0 = dist0;
    tgt_pos1 = dist1;
    /** Buffer is Empty **/
    isEmpty = 1;
    tim_unlock();

    /** Signals an Empty buffer **/
    tim_generate_event();

    /** Set current position as reference **/
    mot_put_sensors_2m(0,0);
    /** Set target position for both motor **/
    mot_new_position_2m(tgt_pos1,tgt_pos0);

******* use tim_wait_sync()********************

    do {
      /** Get controller status for each motor **/
      status0 = mot_get_status(0);
      status1 = mot_get_status(1);

      /** Check is both motors are on target **/
    } while( !((status0 & bitmask) && (status1 & bitmask)) )

    /** Get measured positions **/
    meas_pos0 = mot_get_position(0);
    meas_pos1 = mot_get_position(1);

    /** Coherence check **/
    if( abs(meas_pos0 - tgt_pos0) > POS_TOL ||
        abs(meas_pos1 - tgt_pos1) > POS_TOL )
    {
      tim_lock();
      printf("Warning: Controller critical failure!\r\n");
      mot_reset();
      abs_pos0 = 0;
      abs_pos1 = 0;
      tim_unlock();
    } else {
      /** Protect access to shared data and calculate new position **/
      tim_lock();
      abs_pos0 = abs_pos0 + meas_pos0;
      abs_pos1 = abs_pos1 + meas_pos1;
      tim_unlock();
    }
  };
```

```
}

/** Code for Task 1 **/
static void process_1(void) {
  int32 dist_list0[8] = {};
  int32 dist_list1[8] = {};
  int32 i;

  /** Run continuously **/
  for(;;) {
    for(i=0; i<8; i++)
    {
      /** Check if the buffer is empty **/
      if(!isEmpty)
              tim_wait_event(vIDProcess[0]);

      /** Protect access to shared data and Set new distances **/
      tim_lock();
      dist0 = dist_list0[i];
      dist1 = dist_list1[i];
      isEmpty = 0;
      tim_unlock();

      /** Signal an new available position **/
      tim_generate_event();
    }
  }
}

/** Main entry point **/
int main(void) {
  int32 status;
  static char prName_0[] = "User Process 0";

  /** Reset MOT module **/
  mot_reset();

  /** Install Tasks **/
  INSTALL_NEW_TASK(0);
  INSTALL_NEW_TASK(1);
  return 0;
}
```

This example manages successive movements, from a given position to another. The first process is pooling for new commands and is controlling motors, while the second is simply calculating new positions.

In this simple example, positions are predefined. However, new position calculation may be much more complex, or depend on sensors information so that separated tasks and communication mechanism becomes useful if not necessary.

Tasks are synchronized using event generation mechanism. To avoid unnecessary code execution, tasks are placed in the wait event list so that if the communication buffer is not available, the task is not activated. To make sure no command is missed, an acknowledge system is useful (refer to section 2.1.5 for further information).

The coherence check is useful to detect major controller failures, or eventually other hazardous problems. If the check fails, a message is printed, the MOT module is reseted and the position is set back to 0. This basic reaction is probably not suitable for most applications.

As usual, to avoid data inconsistency, any access to shared variables should be protected using `tim_lock()` and `tim_unlock()`.

2.2.3 PID tuning

2.2.4 New Controller Implementation

The implemented PID loop is good enough for most applications, where motor control algorithm is not the critical feature. Although a new controller can be easily implemented using the PWM control mode and raw sensors feedback. Figure 2.3 gives a block diagram for such a controller.

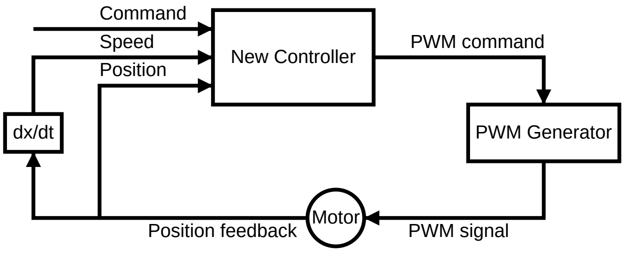


Figure 2.3: Position Control Overview

An important parameter in controller implementation is the sampling period. For very accurate control, a precise and short sampling period is required. Using `tim_suspend_task()` is the only available solution for a periodic task, it might not be robust enough for accurate control, especially if the CPU load is high.

The following code is a generic canvas for a motor control task using a new controller. Using interrupt protection may be necessary, depending on the controller complexity and CPU load.

```
/** Code for Task 0 **/
static void process_0(void) {
  int32 pos0,pos1;
  int32 spd0,spd1;
  int32 pwm0,pwm1;
```

```
  /** Run continuously **/
  for(;;) {
  /** Get motors speed **/
  spd0 = mot_get_speed(0);
  spd1 = mot_get_speed(1);

  /** Get position feedback **/
  pos0 = mot_get_position(0);
  pos1 = mot_get_position(1);

  /** Calculates mew commands **/
  pwm0 = new_controller0(pos0,spd0,command0);
  pwm1 = new_controller1(pos1,spd1,command1);

  /** Apply new commands **/
  mot_new_pwm_2m(pwm1,pwm0);

  /** Suspend task for PERIOD ms **/
  tim_suspend_task(PERIOD);
  }
}
```

2.2.5 Miscellaneous Functions

mot_reset(void)

mot_stop(void)

mot_get_status(uint32 motorNb)

mot_put_sensors_xx(xx)

Set current position as the position reference.

2.3 Sensor Interaction

The Khepera robot is basically using eight infrared sensors to obtain information from its environment. These sensors are both infrared emitter and receptors, they are useful for ambient light intensity measurement and for proximity obstacle detection.

The eight sensors position is displayed on figure 2.4, applications should use the displayed numbers to read a specific sensor.

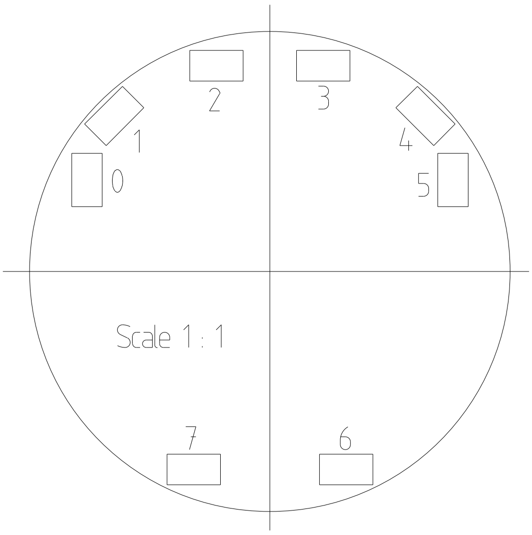


Figure 2.4: Infrared sensors position

2.3.1 Reading Ambient Light

Ambient light intensity is measured with the receptive part of the device. The eight sensors are read sequentially every 2.5ms, so that a full cycle is completed in 20ms.

Several methods are available for an application to read sensors value. For every method, the returned value is always the result of the last measurement made. Therefore a value is never older than 20ms.

The following code use `sens_get_ambient_value()` to read front sensors ambient light measurement every 20ms. As a given sensor value is updated every 20ms, there is no point to use a shorter sampling period. On the other hand, a longer sampling period may be used depending on the application needs.

```
/** Code for Task 0 **/
static void process_0(void) {
```

```
  int32 sens0,sens1;

  /** Run continuously **/
  for(;;) {
  sens0= sens_get_ambient_value(2);
  sens1= sens_get_ambient_value(3);
  printf("Front sensors: %d,%d\n\r",sens0,sens1);

  /** Suspend task for 20 ms **/
  tim_suspend_task(20);
  }
}

/** Main entry point **/
int main(void) {
  int32 status;

  tim_reset();
  sens_reset();
  INSTALL_NEW_TASK(0);
  return 0;
}
```

2.3.2 Reading Reflected Light

Reflected light is measured using both the receptor and the emitter part of the device. During the measurement, a pulse of infrared light is emitted. Reflected light is then calculated as the difference between measured light while emitting and current ambient light value. The eight sensors are also read sequentially every 2.5ms, so that a full cycle is completed in 20ms.

The reflected light value is useful for obstacle detection algorithm. Objects tend to reflect infrared light, the closer they are, the more light is reflected. It is however not possible to directly link reflected light with obstacle distance as results are greatly dependent on environmental conditions and type of detected surface.

The following code use `sens_get_reflected_value()` to read a front sensor measurement. Whenever the value is over a given threshold, a message is displayed.

```
#define THRESHOLD 500

/** Code for Task 0 **/
static void process_0(void) {
  int32 sens0,sens1;

  /** Run continuously **/
  for(;;) {
  sens0= sens_get_reflected_value(2);
```

```
  sens1= sens_get_reflected_value(3);

  if( sens0 > THRESHOLD || sens1 > THRESHOLD )
    printf("Obstacle detected\n\r");

  /** Suspend task for 20 ms **/
  tim_suspend_task(20);
  }
}

/** Main entry point **/
int main(void) {
  int32 status;

  tim_reset();
  sens_reset();
  INSTALL_NEW_TASK(0);
  return 0;
}
```

2.3.3 Filtering IR Sensors

Because of the very noisy and unpredictable nature of IR sensors, using raw measured data is unusually not suitable. Applying some kind of filtering on data is a common practice to enhance detection accuracy.

False or inaccurate detection are often caused by sensors noise and sporadic inconsistent measurement. Several methods are available to reduce false detection problemss, one the simpliest is to average measurements on a given period thus applying basic filtering.

The forgetting factor method is another simple way to eliminate inconsistent measurements. Applications may apply the following calculation, with $\alpha \in [0, 1]$, to sensors value:

$$FilteredValue(t) = \alpha * FilteredValue(t-1) + (1-\alpha) * MeasuredValue(t)$$

The following code is using `sens_get_pointer()` to read front sensors reflected light measurements every 20 ms. The measurement average and the value calculated using the forgeting factor method are displayed every 500ms.

```
#define ALPHA 0.95

uint32 total,nsample,forget;

/** Code for Task 0 **/
static void process_0(void) {
  int32 sens;
```

```
  IRSENSOR * sensor;

  sensor= sens_get_pointer();

  /** Run continuously **/
  for(;;) {
  sens = sensor->oProximitySensors[2];

  tim_lock();
  nsample++;
  total = total + sens;
  forget = ALPHA * forget + (1-ALPHA) * sens;
  tim_unlock();

  /** Suspend task for 20 ms **/
  tim_suspend_task(20);
  }
}

/** Code for Task 1 **/
static void process_1(void) {
  uint32 method1,method2;

  /** Run continuously **/
  for(;;) {
    /** Suspend task for 500 ms **/
    tim_suspend_task(500);

    tim_lock();
    method1 = total/nsample;
    method2 = forget;
    total = 0;
    nsample = 0;
    tim_unlock();

    printf("Sensor Value: %u,%u\n\r",method1,method2);
  }
}

int main(void) {
  int32 status;

  total = 0;
  forget = 0;
  nsample = 0;

  tim_reset();
  sens_reset();
  INSTALL_NEW_TASK(0);
  INSTALL_NEW_TASK(1);
  return 0;
}
```

2.3.4 Using Additional Sensors

2.4 Communication Channels

2.4.1 Serial Communication

2.4.2 Turret Communication

2.4.3 Khepera Extension Bus

4 COMPLEX APPLICATIONS

K-Team SA
1028 Préverenges
CH de Vuasset, CP 111
Switzerland