A Linux kernel driver for the ST-Microelectronics LIS3LV02DL accelerometer for INS-based GPS assistance

Nicholas Angelo Crespi

October 23, 2008

Contents

Preface

As we're heading towards integrated, embedded portable devices, manufacturers are quickly adding new features to their products: one of these, once prerogative of specialized navigation devices, is a GPS receiver. These devices have the challenge to be useful navigators even when their GPS receivers are prevented from attaining a fix on enough satellites. It happens often in urban canyon settings, as tall buildings and concrete walls hinders the GPS signal reception. When this happens the GPS receiver must be *assisted* with information coming from other sensors in order to synthesize the current position.

Chapter 1

Introduction

1.1 Global navigation satellite system

Global Navigation Satellite System is the standard generic term for satellite navigation systems that provide autonomous geo-spatial positioning with global coverage. A GNSS allows small electronic receivers to determine their location (longitude, latitude, and altitude) to within a few metres using time signals transmitted along a line of sight by radio from satellites. Receivers on the ground with a fixed position can also be used to calculate the precise time as a reference for scientific experiments.

1.1.1 Global Positioning System

As of 2007, the United States NAVSTAR Global Positioning System (GPS) is the only fully operational GNSS. It uses a constellation of between 24 and 32 medium earth orbit satellites that transmit precise microwave signals, that enable GPS receivers to determine their current location, the time, and their velocity (including direction). GPS was developed by the United States Department of Defense.

A GPS receiver calculates its position by carefully timing the signals sent by the constellation of GPS satellites high above the Earth. Each satellite continually transmits messages containing the time the message was sent, a precise orbit for the satellite sending the message (the ephemeris), and the general system health and rough orbits of all GPS satellites (the almanac). These signals travel at the speed of light (which varies between vacuum and the atmosphere). The receiver uses the arrival time of each message to measure the distance to each satellite, from which it determines the position of the receiver, using trilateration techniques. The resulting coordinates are converted to more user-friendly forms such as latitude and longitude, or location on a map, then displayed to the user.

It might seem that three satellites would be enough to solve for a position, since space has three dimensions. However, a three satellite solution requires the time be known to a nanosecond or so, far better than any non-laboratory clock can provide. Using four or more satellites allows the receiver to solve for time as well as geographical position, eliminating the need for a very accurate clock. In other words, the receiver uses four measurements to solve for four variables: x, y, z, and t. While most GPS applications use the computed location and not the (very accurate) computed time, the time is used in some GPS applications such as time transfer and traffic signal timing.

GPS problems

Since GPS signals at terrestrial receivers tend to be relatively weak, it is easy for other sources of electromagnetic radiation to desensitize the receiver, making acquiring and tracking the satellite signals difficult or impossible.

Solar flares are one such naturally occurring emission with the potential to degrade GPS reception, and their impact can affect reception over the half of the Earth facing the sun. GPS signals can also be interfered with by naturally occurring geomagnetic storms, predominantly found near the poles of the Earth's magnetic field. GPS signals are also subjected to interference from Van Allen Belt radiation when the satellites pass through the South Atlantic Anomaly.

GPS signals are also degraded by different artificial sources. In automotive GPS receivers, metallic features in windshields, such as defrosters, or car window tinting films can act as a Faraday cage, degrading reception just inside the car. Man-made EMI (electromagnetic interference) can also disrupt, or jam, GPS signals.

Urban environments with streets cutting through dense blocks of structures, especially skyscrapers, have a great impact over GPS signals reception. Unfortunately, this happens when the need for localization is at its highest peak. To solve this problem, a receiver should rely on another method of positioning while the GPS signal is unavailable.

1.2 GNSS augmentation

GNSS Augmentation involves using external information, often integrated into the calculation process, to improve the accuracy, availability, or reliability of the satellite navigation signal. There are many such systems in place and they are generally named or described based on how the GNSS sensor receives the information. Some systems transmit additional information about sources of error (such as clock drift, ephemeris, or ionospheric delay), others provide direct measurements of how much the signal was off in the past, while a third group provide additional navigational or vehicle information to be integrated in the calculation process.

The augmentation may also take the form of additional information being blended into the position calculation. Many times the additional navigation sensors operate via a different principle than the GNSS and are not necessarily subject to the same sources of error or interference. The additional sensors may include:

- Automated Celestial navigation systems;

- Simple Dead reckoning systems (composed of a gyro compass and a distance measurement);
- Inertial Navigation Systems.

Among all these methods, the latter is rapidly gaining accuracy and cost effectiveness as the digital sensor technology is maturing. Furthermore, the development of motion sensors built built with the MEMS[1] technology enabled the embedding of navigation systems in portable devices.

1.2.1 Inertial navigation with GPS and accelerometers

GPS systems have some problems when working in closed environment or in urban areas, where the need of localization is highest. An inertial navigation system could be used in concert with GPS positioning when the signal is unavailable.

Inertial navigation systems using accelerometers is based on the numerical integration of accelerations coming from the sensors. Accelerometers measures linear acceleration among some given axis: to obtain information about the angular velocity of the device we need either a couple of accelerometers in a shifted-axis configuration, or a set of gyroscopes and an accelerometer. If there's only one accelerometer available, we lose information about the rotation of the device: however, it the device is kept still we could still measure its motion.

Computing motion from acceleration is accomplished through a simple double integration. Given our acceleration $a(t)$ and a time $t_0 < T$ when the device wasn't in motion, we can compute the speed function $s(t)$ as:

$$s(t) \;=\; \int_{t_0}^{t} a(t)\,\mathrm{d}t\,.$$

We can now obtain the position $p(t)$ as the integral of the speed function computed before:

$$p(t) \;=\; \int_{t_0}^{t} p(t)\,\mathrm{d}t\,.$$

It's worth nothing that the error term caused by the sensor (composed of uncertainty, quantization error, nonlinearity, ...) is magnified by the double integration. Another error source arises from the numerical integration algorithm chosen. Anyway, the distance covered without the GPS fix should be small, so the estimation error imposed by the numerical integration shouldn't pose a big problem.

[1]Microelectromechanical systems

Chapter 2

Requirements And Constraints

The project's goal is to write a linux kernel driver for the LIS3LV02DL MEMS sensor manufactured by ST and write a simple daemon, *assistd*, which provides position updates based on the sensor's readings.

As this project is academic in nature, most of requirements have been stated in advance by the project supervisor. However, further meetings with the Nomadik team at ST provided some additional requirements as well as some constraint over the driver's interface.

2.1 Driver requirements

The driver must be written to serve several applications, each with it's own set of requirements:

GPS assistance: as stated before, one of the goal is to implement a small daemon that assists gpsd by providing position updates synthesized from the accelerations. To accomplish this the driver must provide samples with its greatest accuracy and with a frequency of 30 Hz or better. Furthermore, the output interface must be fast compared to the output frequency and easily accessible from userspace. Finally, the driver must provide either its updating frequency via a configuration interface or it should block readings from cycle to cycle. We'll discuss more about the assistd requirements later.

Gesture support: this application allows to perform gestures by simply moving the device around. To accomplish this, the driver must provide samples at least at 30 Hz, with no constraint on accuracy. Moreover, the acceleration samples must be equally spaced in time. Similar to GPS assistance, the output interface must be accessible, fast and synchronized (i.e.: it should block reads until it has new data).

Joystick: this set of requirements is focused towards allowing to use the device as a joystick by moving it around. To accomplish this, the driver must provide samples at 30 Hz or more, with no hard constraints on accuracy. Again, it must also have a fast, accessible output interface that provides its own synchronization.

To sum up, we need to write a device driver that could sample at least at 30 Hz, with high sensitivity (when asked to), it should provide a fast, accessible output interface that synchronize applications by blocking the readings on the interface until new samples are available.

As these applications are mutual exclusive, there's no need to care about multiple application reading the output interface, although this feature could be a nice addition.

2.1.1 Hardware platform constraints

Embedded systems development faces important requirements and constraints arising from the specific platform that will be used. As the project is focused towards developing software for Nomadik, an embedded portable device, we must investigate carefully platform-specific constraints.

For this project, the platform used will be the Nomadik NHK-15 r3.1 reference board from ST Microelectronics. It's based on the STn8815 SoC that includes an ARM processor (ARM926EJ), two I2C bus adapter (named I2C-0 and I2C-1), smart graphics acceleration, general purpose I/O interface and so on.

The MEMS sensor that we'll use, the LIS3LV02DL, has an adjustable sensing frequency between 40 Hz and $2,56$ KHz and an adjustable fullscale of ± 2 g or ± 6 g. It also supports data-ready interrupt generation, direction detection interrupt generation with adjustable conditions and thresholds and free-fall wakeup interrupt generation, again with adjustable conditions and threshold. The sensor supports either I2C or SPI bus, but on the NHK-15 r3.1 board it's configured to use the I2C interface, and it's connected to the SoC through the I2C-0 adapter. Its interrupt pin is connected via GPIO port 82. For more information about this sensor, you can find its datasheet at www.st.com.[1]

2.2 Assistd Requirements

The other goal of the project is the development of assistd, a GPS assistance daemon. For the same reason of the driver, the project supervisor provided a list of requirements for the daemon to be met.

Assistd must be a UNIX daemon; it have to base its configuration on an external configuration file, which can be provided as an argument (in particular, the location of both the output interface and the control interface of the driver must be provided in that file). The daemon should accept a command line

[1]http://www.st.com/stonline/products/literature/ds/12094/lis3lv02dl.htm

parameter that specifies the location of the configuration file: if it's not present, it should look for the configuration file in the `/etc` directory.

The primary purpose of the daemon is to provide position updates. A position update is defined as an estimate of the distance covered by the device from the last time an update was requested to the time of the current request, measured in meters.

The requirements associated to the daemon can be divided in two subsets: one associated with the internal implementation and one associated with the output interface.

Talking about the internal implementation, assistd must implement some sort of quadrature rule algorithm in order to obtain position updates from the sensor readings. Dead reckoning systems based on accelerations are known to diverge quickly, because of double error integration), hence there's no strict constraints on the approximation error introduced by the integration method as it's supposed to be conveniently small. However, if several integration method are individuated, the method with the least approximation error should be chosen. On the other side, the daemon should take samples at least at the sensor's minimum output frequency (40 Hz for LIS3LV02DL), so it must implement an integration method that's fast enough to keep pace with that.

The output interface of the daemon should be easily accessible from other daemons (such as gpsd). It should also have a mechanism that senses when another application reads the stored positions and thus reset them (in order to provide position updates). It should also provide a method of clearing the stored speeds.

The structure of the position updates is also subject of some requirements. It must be a three double-precision floating point vector, each containing the position update along a single axis. The axis ordering for the updates can be depicted as below.

$$\begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}$$

2.3 Software platform constraints

The version of linux that will be used for the development is STlinux 2.3, which contains a 2.6.20 kernel patched with platform-specific code. As the linux API changes from version to version, knowing that you'll work on a specific kernel version will ease you task a lot, because there won't be the need to read through different trees in order to see what's changed and what's not. The software toolchain for cross-compiling the software is included in the STlinux 2.3 release, as well as a script for setting the appropriate environment variables.

Chapter 3

Design

This chapter deals with the translation of the previously gathered requirements into a design for both the driver module and the integration daemon. It introduces the possible solutions that could address the requirements and the drawbacks associated with them; then, it shows the reason why a specific solution has been preferred. Finally, the architecture of the two pieces of software will be discussed.

It's worth noting that this chapter is written to resemble a single design phase, as in the waterfall development model, even though the project was based on an iterative development model. It's an expedient that has been adopted in order to provide an organic report of the design phase, instead of a list of gradual improvements applied on an initial bare design.

3.1 MEMS driver

The requirements gathered in the previous phase are primarily oriented at how the device must be configured to operate and what performance must met. The few architectural constraints gathered in the previous phase address the output interface performance and the configuration interface.

3.1.1 Configuration interface

The standard configuration interface provided by the linux kernel is sysfs. It's a virtual file system that is used to export information about devices and drivers from the kernel device model to userspace, and is also used for configuration.

Its theory of operation is straightforward. For each object in the driver model tree a directory in sysfs is created. For device drivers there's the possibility to create attributes, which are simple files: the rule is that they should only contain a single value and/or allow a single value to be set. These files show up in the subdirectory of the device driver respective to the device. It's also possible to create attribute groups, subdirectories filled with attributes.

Based on this properties, we decided to use sysfs to create a configuration interface for the device.

3.1.2 Output interface

For this interface we have a few options, as the linux kernel provides several methods to export data to userspace.

Sysfs. As stated above, sysfs can be used also as an output interface, by configuring a read-only attribute and providing a show function. This method is slow and doesn't provide any native locking mechanism, but it's a nice addition for troubleshooting purpose; plus it doesn't require much effort to implement.

Character Device. A character device is a special file related to a device that transmits data one byte at a time. The linux kernel provides a simple implementation of a character device: it provides a structure that holds a collection of pointers to functions that will be called when the corresponding operation will be performed on the file (open, read, write, seek, close, flush and so on), plus two API to register and remove it.

This approach has several advantages over sysfs: the character device file doesn't need to be opened and closed at every read, saving a considerable amount of time; it can also export multiple values over a single file, so they can be read simultaneously. However the device returns data in byte-sized chunks (c type `char`), so a software that have to read acceleration data (three *signed word*) from this interface must employ its own conversion routine.

Like sysfs, a character device doesn't provide native locking mechanism, although it's possible to implement a blocking char device using kernel APIs like `wait_event` and `wake_up`.

Memory Mapping. The most famous memory mapping API is `mmap`, which allows the mapping of device memory directly into a user process' address space, providing a user programs with direct access to device memory.

It's a very fast output interface, but it doesn't provide a locking mechanism and lacks a method to link custom functions to operations performed on the memory, making it hard to implement a locking mechanism.

Input Interface. The linux input subsystems provides a simple method of exporting data coming from input devices such as keyboards, joysticks and mices to userspace: the event interface. Each event interface is a special file that provides synchronized data coming from the device. The data is pushed in the event queue in the form of *events* from a kernel driver that have registered an input interface by means of a standard set of API.

Each event is composed by three values: a *type* that discriminates between several predefined event types (absolute movements, relative movements, keypress, and so on) a *code* that discriminates between events with the same type (i.e.: linear movement versus angular movement) and a *value* that quantify the event. Each event also carries a `timespec` structure containing the time-stamp of the event.

An input interface provides its own method of locking. A read performed over the event special file blocks if there's no event ready in the queue.

> Furthermore, the presence of a fine-grained time-stamp is useful when the device is asked to generate free-fall or direction detection signals.

For this driver, we decided to use a mixture of three of these interfaces.

We decided to use sysfs output interface for accelerations, as it's very graphical and easy to use: it requires a simple `cat` to be performed on the attribute. It has helped us while trying different device configurations. It is meant only for debugging and configuring purposes, hence it does not block waiting for any interrupt: it simply reads the data from the sensor and print it.

Another interface used in this driver is a character device file, with *major number* 190. This interface can be used only with data-ready signal generation enabled, as it blocks any reading performed upon it until the sensor generates a new data-ready interrupt; then it return six bytes, containing the three accelerations read from the sensor, measured in *LSB/g* (based on the sensor configuration, 1 LSB can vary from 1/1024 g to 1/340 g). The reader software must employ its own conversion mechanism in order to convert this data in a convenient measurement unit. To disambiguate which of the three words represents which acceleration, an ordering convention has been formalized: it mandates that the first acceleration is relative to the X axis, the second is relative to the Y axis and the third is relative to the Z axis.

Finally, this driver implements an event input interface. This input interface can be used only when data-ready signal generation or interrupt generation are enabled, as the events can be generated only when the driver is in interrupt-driven mode. The events type returned are `EV_ABS` for acceleration data (data-ready enabled) or `EV_REL` for free-fall and direction detection data (interrupt enabled).
If the driver is in data-ready mode, three events are generated at every interrupt corresponding to the codes `ABS_X`, `ABS_Y` and `ABS_Z`: their value it's the acceleration read from the sensor, in the same format that's used by the character device interface. Then, a synchronization event is generated.
If the driver is in interrupt mode, six events may be generated: event codes `REL_X`, `REL_Y` and `REL_Z` indicates free-fall interrupt, while event codes `REL_RX`, `REL_RY` and `REL_RZ` indicates direction detection interrupts. The value associated to these six events can be $+1$ or -1, with different meaning based on which kind of interrupt has been intercepted.
A free-fall event with value $+1$ is generated whenever the absolute value of the acceleration along any given axis exceed a preset threshold, while an event with value -1 is generated whenever the absolute value is lower than the threshold.
A direction detection event with value $+1$ is generated whenever the acceleration along any given axis exceed a hysteresis region defined by two thresholds ($+THSI$ and $+THSE$), while an event with value -1 is generated whenever the acceleration along any given axis is lower than a hysteresis region defined by the same thresholds inverted ($-THSI$ and $-THSE$). It's worth noting that, while direction detection events along an axis are always mutually exclusive, based on how the free-fall configuration register is set both positive and negative free-fall events might occur.

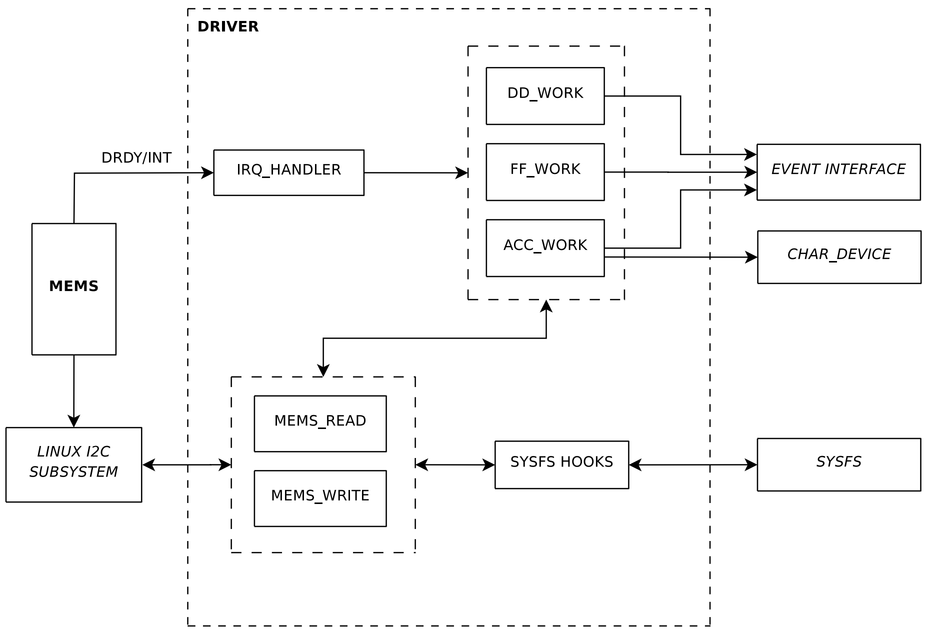

Figure 3.1: The MEMS driver architecture block diagram.

3.2 MEMS driver architecture

Figure 3.1 shows the architecture of the sensor driver, as it is implemented. This driver is based on the I2C device driver model, hence it contains a number of functions that the I2C infrastructure needs in order to probe, initialize, attach and detach it.

The two interface functions `mems_read` and `mems_write` wrap the corresponding I2C transferring functions in order to make the driver independent from minor API changes. These functions handle all the communication to and from the device.

The sysfs hooks block is a group of procedures exported as sysfs attributes, which in turn forms the configuration interface of the driver. They handle the conversion from sensor values to human readable values and vice-versa. For simplicity, in this first implementation of the driver all the attributes are grouped into a single `sysfs_group` which is exported as a directory named *nmdkmems*.

When the driver is inserted in the kernel an interrupt handler is initialized, linked to the GPIO pin where the sensor is attached to; the driver initializes also three work function (`acc_work`, `DD_work` and `FF_work`) that perform the operation that has been requested with the interrupt. When data-ready signal generation or interrupt generation has been enabled they begin to process interrupts coming from the sensor, feeding the output character and event interfaces with data.

The two output interfaces exported by the driver are a character device and an event input interface.
The character device created has a *major number* of 190, and its special device

can be created by the superuser by executing

```
mknod /dev/nmdkmems c 190 0,
```

or by configuring `udev` to dynamically create it at insertion.
the input interface is dynamically linked by the kernel to an event interface. A list of input interfaces with the respective event handlers can be found in the file

```
/proc/bus/input/devices.
```

3.3 Assistd

In this section the requirements specified in the previous chapter about assistd will be addressed. They constrain the architecture of the daemon more than those related to the driver as well as the functions it is required to perform.

3.3.1 Reading from the kernel driver

The MEMS driver, whose architecture has been already discussed in the previous section, provides three output interface: sysfs, character device and input interface. As sysfs provides accelerations only for debugging and configuring purposes, the choice for assistd's input interface is narrowed down to two.

The character device offer a low-overhead interface, exporting only six bytes at a time without any other information (other than the ordering constraint defined above); it also provides its own locking mechanism, freeing the daemon of the burden of having a timer waking it up.

The same properties also holds for the input interface, but it exports much more information to userspace than the character device (though it's simpler to discriminate accelerations, as every event have it's own type and code). This could be useful if the locking mechanism isn't working properly and the daemon have to get time information from the events timestamp in order to integrate properly, but for now it's only a time waste.

For this reason, the reading interface used by assistd will be the character device exported by the driver.

3.3.2 Integration

When the driver was almost complete, a set of acceleration samples has been acquired in order to evaluate graphically the different numerical integration methods[1] available for implementation.

The samples acquired were about a single axis movement of about 30 *cm*, covered at variable speeds and accelerations. The log were then integrated manually using different numerical integration algorithms in order to evaluate which one was better for the daemon.

[1]the results are presented in the next chapter.

Integration rules

The MEMS sensor provides us with equally-spaced samples of the acceleration function $a(t)$: the obvious choice for the integration is thus an algorithm implementing one of the Newton-Cotes formulas, which take advantage of the homogeneous spacing. Newton-Cotes formulas provides accurate estimation of the integral if the sampling frequency is conveniently small, and the higher the degree of the formula the smaller the error is. However some rule's error term increase if the integrand function is highly oscillatory.

From the aforementioned family we chose two of the simpler rules: the *trapezoid rule* and the *Simpson's rule*. We decided to take these two because one of the requirements for the integration process addresses its speed; the other point is that being low in degree, they update their estimate with a higher frequency than higher degree rules (up to 1/4 of the sampling frequency if we use double Simpson's rule integration).

The **trapezoid rule** is the simplest among all the Newton-Cotes formulas. It works by approximating the region under the graph of the integrand function with a trapeze and calculating its area. It follows that:

$$\int_a^b f(x)\,\mathrm{d}x \;\approx\; (b-a)\,\frac{f(a)+f(b)}{2}\,.$$

The error term associated with the rule is

$$e(h) \;\approx\; -\frac{h^3}{12}\,f^{(2)}\,(\xi)\,.$$

This rule has some advantages over Simpson's rule when the integrand function is not twice continuously differentiable, and gets extremely accurate when periodic functions are integrated over their periods.

The **Simpson's rule** is a Newton-Cotes quadrature rule of degree 2, derived by replacing the integrand with the quadratic polynomial build over three equally spaced points $f(a)$, $f(b)$ and $f(c)$ by using Lagrange polynomial interpolation. Easy calculations derives that

$$\int_a^c P(x)\,\mathrm{d}x \;\approx\; \frac{(b-a)}{6}\;[\,f(a)+4f(b)+f(c)\,]\;.$$

The error term associated with the rule is

$$e(h) \;\approx\; -\frac{h^5}{90}\,f^{(4)}\,(\xi)\,.$$

Simpson's rule gains an extra order of the error term because the points at which the integrand are evaluated are distributed symmetrically in the interval $[a,c]$. Simpsons's rule has some advantages over trapezoid rule when the integrand is "smooth" enough, i.e. when it is twice continuously differentiable and it's not highly oscillatory: if this is not true, Simpson's rule may give very poor results.

Results

To obtain position estimates from accelerations samples, a double numerical integration is required. As we have to test two rules, we have 4 different experiments to carry on: Trapezoid-Trapezoid, Trapezoid-Simpson's, Simpson's-Trapezoid and Simpson's-Simpson's.

The results of the experiments show that the approach with the least error in all of the different samples group is, surprisingly, Simpson's-Trapezoid. It's surprising, as the acceleration function sampled by the sensor is very oscillatory, and the maximum frequency at which we could sample is limited by the overhead of the I2C adapter device driver to 40 Hz. Still, we chose this as our method of numerical integration.

3.3.3 Output interface

The output interface that will be implemented in assistd should be fast, easily accessible and must provide a mechanism of clearing the position updates once read.

For this interface linux offers two valid alternatives:

Shared Memory: one of the simplest interprocess communication services, a shared memory is a memory segment that is shared between two or more processes, as if they all called `malloc` and were returned pointers to the same actual memory. It is the fastest form of interprocess communication because all the processes share the same piece of memory, and its access time is as fast as accessing a process's nonshared memory. The shared memory segment isn't synchronized, so it must provide its own method of synchronization (typically with a mutex inside the segment); also, it's not possible to know whether or not someone has read the memory, so the reader must set some flag in the shared memory to let the daemon know the updates has been read.

Socket: a socket is a bidirectional communication device that can be used to communicate with another process on the same machine (or with a process running on other machines). They have two different method of communication: the first one, *connection*, guarantees delivery and ordering; the second one, *datagram*, which doesn't. As it's substantially a buffer managed by the kernel, reading and writing over a socket requires the invocation of two API and the use of some sort of communication protocol between the two processes.

Both of the candidates can be used to implement the output interface of assistd: a shared memory with a flag and a mutex, and a socket with a simple communication protocol.

However, we think that speed is a crucial factor for a software developed for embedded devices, so we choose to implement a shared memory segment for our daemon.

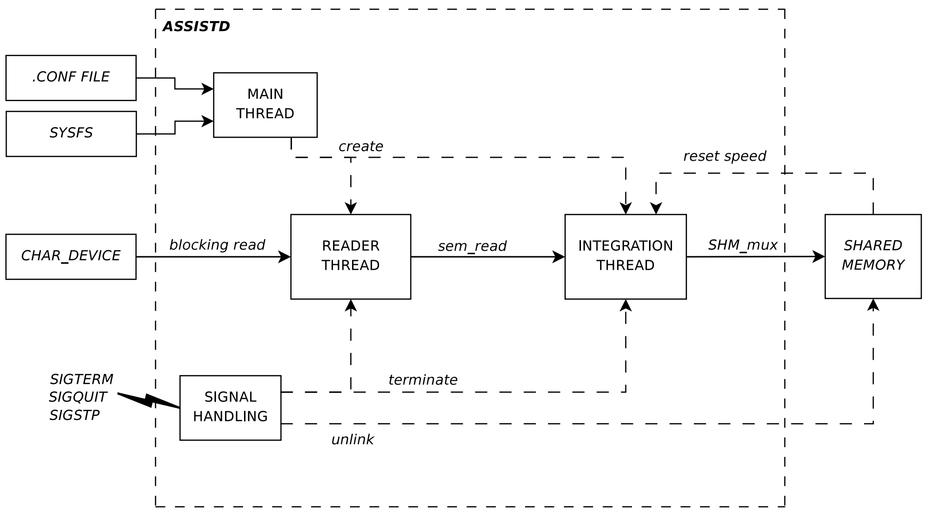

Figure 3.2: *Assistd* architecture block diagram.

3.4 Assistd architecture

Figure 3.2 shows the architecture of the daemon, as it is implemented. The main thread loads the configuration from the file, configures the driver by using its sysfs interface, initializes the internal structures and the shared memory segment, spawns the reader and the integrator thread, protects the daemon from `SIGTERM`, `SIGSTP` and `SIGQUIT` signals and goes to sleep.

The reader thread opens the character device exported by the driver. At every cycle the thread reads from the character device (blocking if no data is available) and put it in a buffer shared with the integrator thread. Finally, he post the semaphore `sem_read` over which the other thread blocks, waking it up.

The integrator thread initializes itself and then wait on `sem_read`. When the reader thread will post the semaphore it will wake up, copy the accelerations from the shared internal buffer and then evaluate if it has enough data to update its speed and position estimates. If it's the case, the thread will perform the numerical integration (as the integration from speed to position is performed using trapezoid rule, each acceleration integration leads to a speed integration) to obtain new position estimates; then, it will block the shared memory for writing and it will look for the read and the reset flags in the shared memory block. If any of these two are set it will reset the respective field, then it will write the new position updates in the segment.

If the daemon receive one of the three signal against which it's protected, it will gracefully stop execution; otherwise, a manual restart will be required in order to clear the data pending from the previous execution.

3.5 Testing

While testing both the driver and the daemon we found several bug: some of them have considerably hindered the utility of the project. Here we present some of the most critical unresolved bugs.

- The I2C adapter driver adds an excessive overhead when communicating with the device. We performed a test with a high-performance clock in order to assess how much time a single byte read or write were taking: it turns out that a single read or write takes from 6 ms to 7 ms. Combined with the bug described below, this bug didn't allowed us to use the sensor to its full potential.

- The I2C adapter driver doesn't allow the reading of multiple bytes from contiguous registers. The read API provided allow only for a single byte to be read at a time, thus considerably increasing the reading time of the accelerations (6 bytes). This bug, combined with the previous one, allows us to read only one acceleration (two bytes) at the lower frequency at which the sensor can be set (40 Hz), hindering considerably the usefulness of the sensor.

- There's a minor bug in the GPIO driver implementation of Nomadik (probably in the `arch/arm/mach-nomadik/gpio.c` source file). When our sensor driver is inserted the first time, it registers an interrupt handler with GPIO port 82; when the driver is removed, our driver calls `free_irq` in order to free the interrupt. Something happens here as the interrupt is not freed completely: the execution of `cat /proc/interrupt` shows that the interrupt has been successfully cleared, while the execution of `cat /proc/gpio` results in a kernel fault.

- Sometimes reading accelerations from the sysfs interface leads to a spurious value of ±255. It should not pose a big problem as sysfs it's meant for debugging and configuration only, nonetheless the bug has been forwarded to the Nomadik team.

Chapter 4

Experimental Results

In this chapter we present some experimental results of our work. Our goals were to find a sensor configuration that was good enough for all the applications that required the sensor data and to test both the driver and the userspace daemon in order to see if they were working correctly together.

4.1 The experiment

When the driver implementation was almost complete, we conducted some test in order to choose which integration method we should use for the daemon, and to see if the acceleration function read from the sensor was close to the real acceleration imparted to the board.

We set up a simple experiment. The board was set up with a fullscale of $\pm 2\ g$, the high-pass filter enabled for the acceleration signals with a frequency cut-out selection of 512, all the three axis disabled and data-ready signal generation enabled. With this configuration, the accelerometer's sensitivity is at its maximum.

We slid the board along a ruler letting it cover a fixed length, while we were logging the acceleration values coming from the device. In this manner we could focus on accelerations coming from one axis at a time, leaving the other two disabled. Then, we changed the fullscale to $\pm 6\ g$, and proceed to take a set of samples.

To post-process the samples we put them in a spreadsheet and we plot them, in order too see if the sensor was working correctly. Then, we applied different numerical integration method, in order to choose the one with the least integration error. Finally, we tried to apply different divisor to the acceleration read from the sensor, in order to assess which divisor led to cut the noise in the data without dampening the actual acceleration.

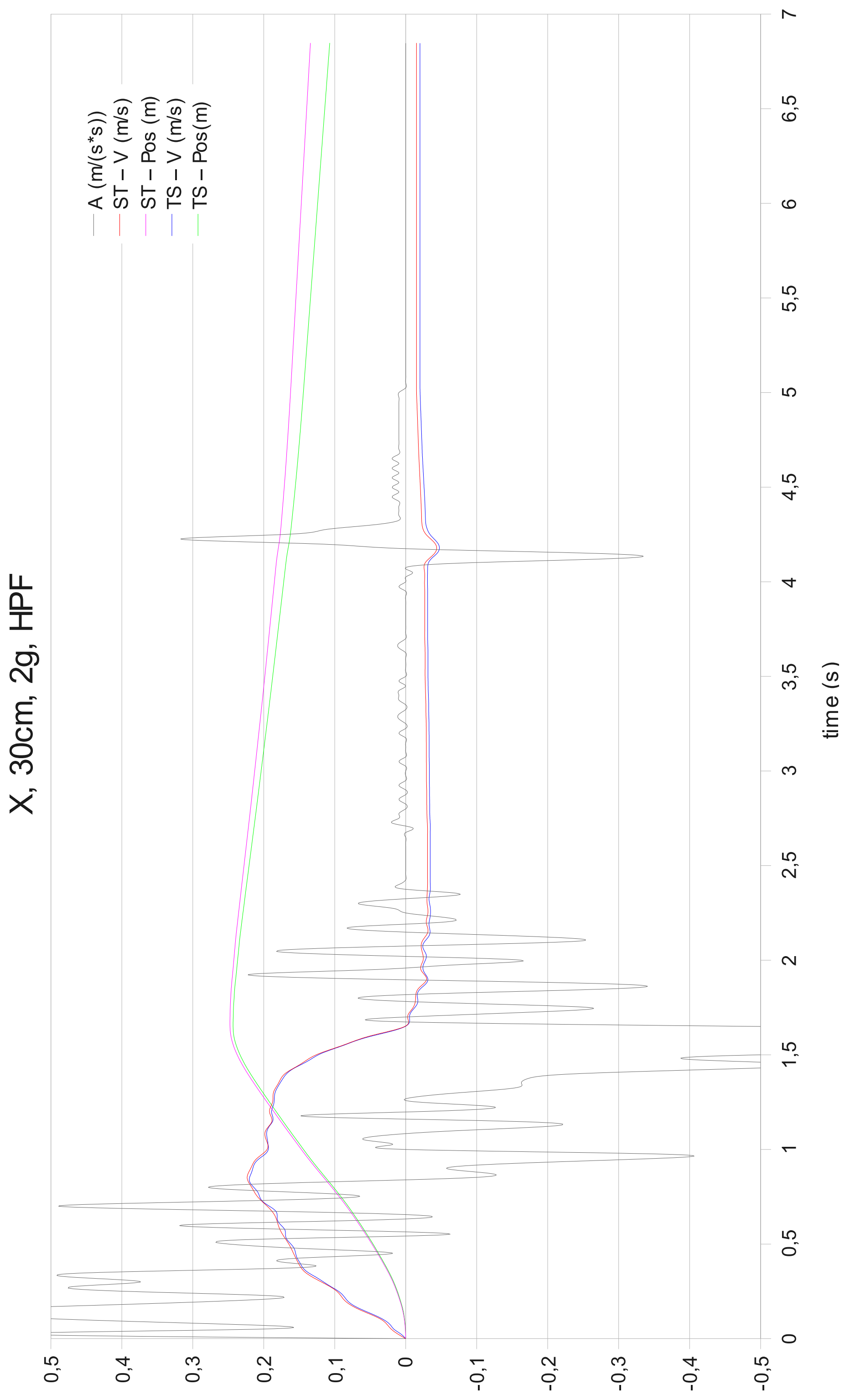

Figure 4.1: Different Integration methods applied to a set of samples taken from a 30cm motion along the X axis, with a fullscale of 2 g and with HP filter enabled.

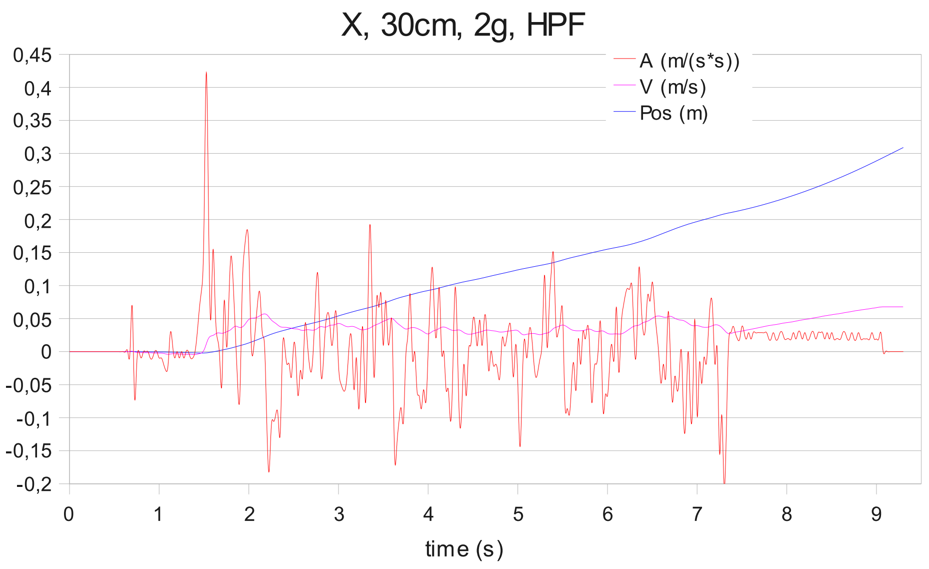

Figure 4.2: Diverging speed and position estimates caused by inadequate sampling frequency.

4.2 Integration method

Figure 4.1 shows two different numerical integration methods applied to a set of samples coming from an experiment. The ST and TS in the legend refers to the order of application of the two quadrature rules: ST means that Simpson's rule was applied to the raw data in order to obtain the speed function, and then the trapezoid rule was applied to the result in order to obtain the position estimates; vice-versa for TS.

The graph shows that ST leads to a final speed that is less than the one computed through TS, hence giving a position estimate that degrades less than the other. Furthermore, the graph shows that the computed speed isn't zero and the position estimate degrades over time;furthermore, it is far less than 30 cm.

inerWhile the first issue can be corrected by thresholding the signal coming from the sensor, for example dividing every reading by a predefined scale value, the second issue is much harder to correct.

4.3 Sampling frequency

Figure 4.1 shows that, even when integrating with ST, the final position isn't even near to the real movement of 30 cm. This effect even more pronounced in other samples we took, for example the one graphed in figure 4.2.

This error is supposed to be linked to the sampling frequency of the sensor: it's possible that the frequency we're struck to[1] doesn't allow to acquire the

[1]see section 3.5 for further info.

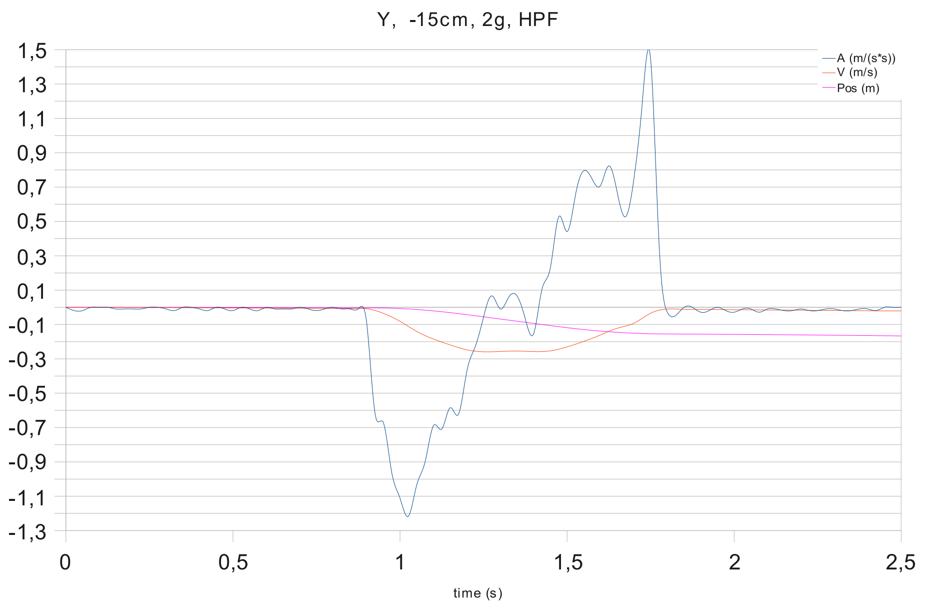

Figure 4.3: Oscillating final speed with a more accurate position estimate. the position at $1,8\ s$ is $-0,16\ m$.

acceleration function properly.

Surprisingly, things got much better when we took samples by moving along the positive X axis or the Y axis. For example, look at figure 4.3: the final speed is near zero, and the position estimate is near the nominal movement value of $15\ cm$. We can correct the speed behaviour by thresholding, as shown in the next section.

4.4 Thresholding accelerations

One of the simplest method for removing error (especially additive Gaussian white error) from a signal is by dividing each sample by a predefined divisor.

The simplest method of performing this is to shift the register containing the value right by a predefined number of bits, dividing the register by an increasing power of two for each bit shifted (it work even if the register has a two's complement's value, though the sign bit must be copied in the added MSB).

We experimented with different thresholds over different set of samples to see if there was a divisor that could have the maximum performance in cutting the rumor while still preserving the acceleration impressed to the sensor.

For example, figure 4.5 shows integrated speed from the same set of samples, divided by various power of two (from 1 to 4), while figure 4.5 shows the position estimates computed from the calculated speeds. You can note that dividing the acceleration by 8 leads to a perfect final speed, while the position isn't much

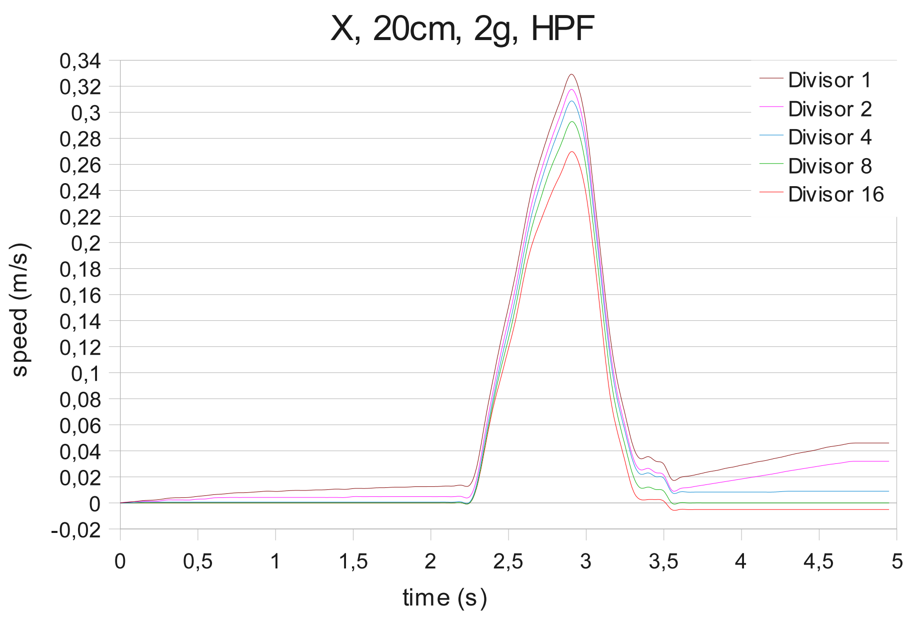

Figure 4.4: Computed speeds from divided accelerations.

accurate. In contrast, dividing by 4 leads to a slight positive speed, but the position estimate is more accurate.

The next series of graphs picture the same set of samples with various thresholds applied (dividing each samples by 1, 2, 4 and 8), together with the computed speed and position. This case is somewhat unfortunate, as even the best estimate never reach the nominal movement of 15 cm (it stops at 12 cm). However, let's see how different divisor influences the estimate.

The original acceleration is graphed in figure 4.6. you can notice how the final speed is lower than zero and gradually increasing from spurious oscillations of the samples: this bias pulls down the position estimate, resulting in a degraded estimation.

A simple division by two, shown in figure 4.7, leads to a more stable speed. The position estimate is more stable, and it's nearer to the nominal movement value of 15 cm. However, there's still some oscillation visible at $2,5$ s: if it's very short however, and doesn't influence the computed speed.

Dividing the original acceleration by 4 eliminates the spurious ripple at $2,5$ s, giving a even more accurate estimate.

Dividing by 8, as shown in figure 4.9, cuts some of the acceleration. The speed is now even closer to zero, and the estimate is as accurate as the one graphed in figure 4.8. This result is thus the most stable between the four analyzed here.

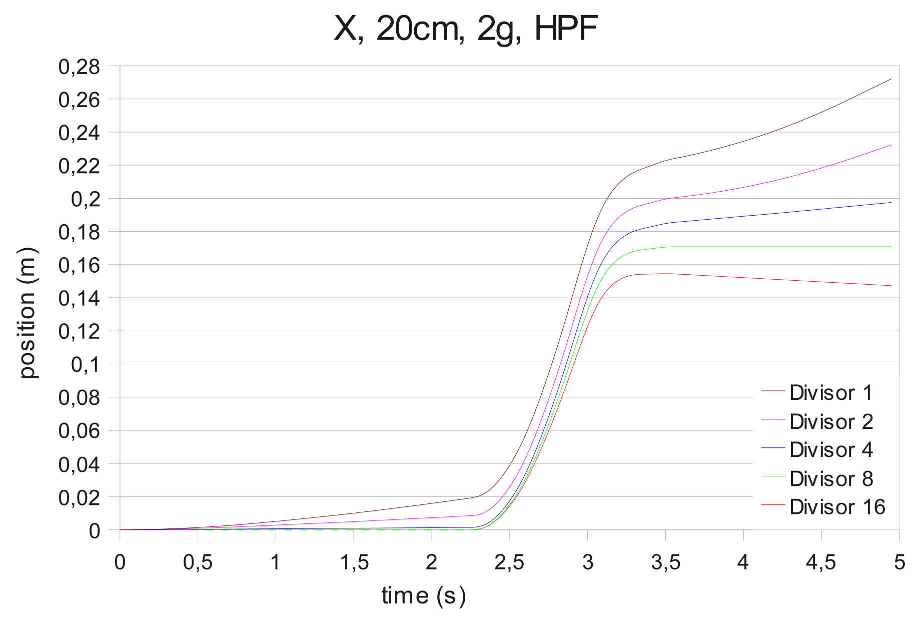

Figure 4.5: Position computed from speeds graphed in figure 4.5.

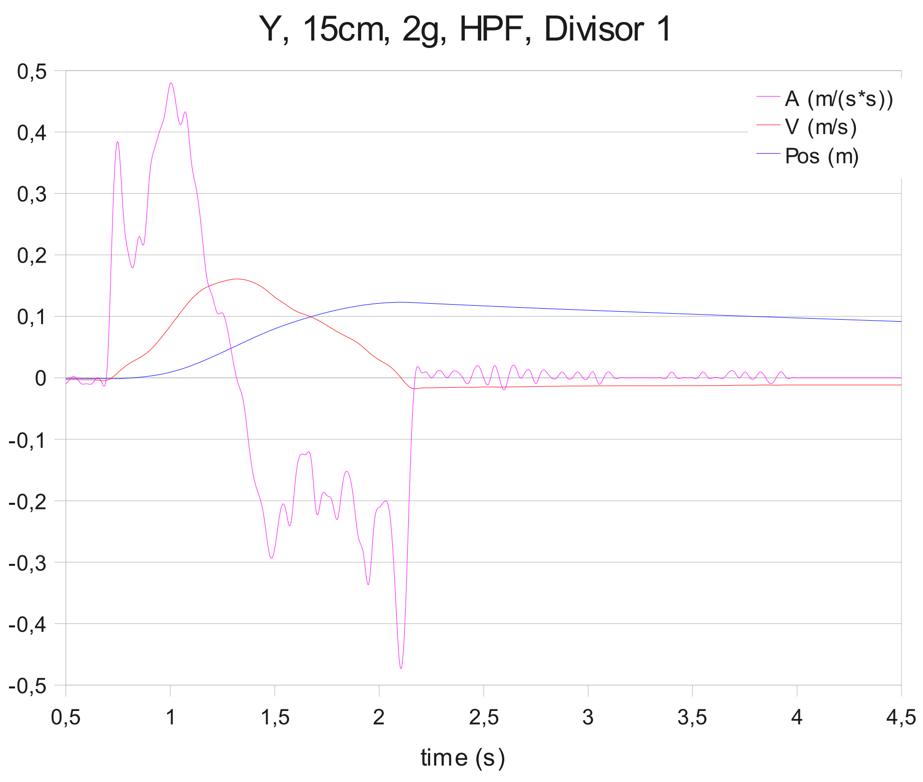

Figure 4.6: Original acceleration, speed and position

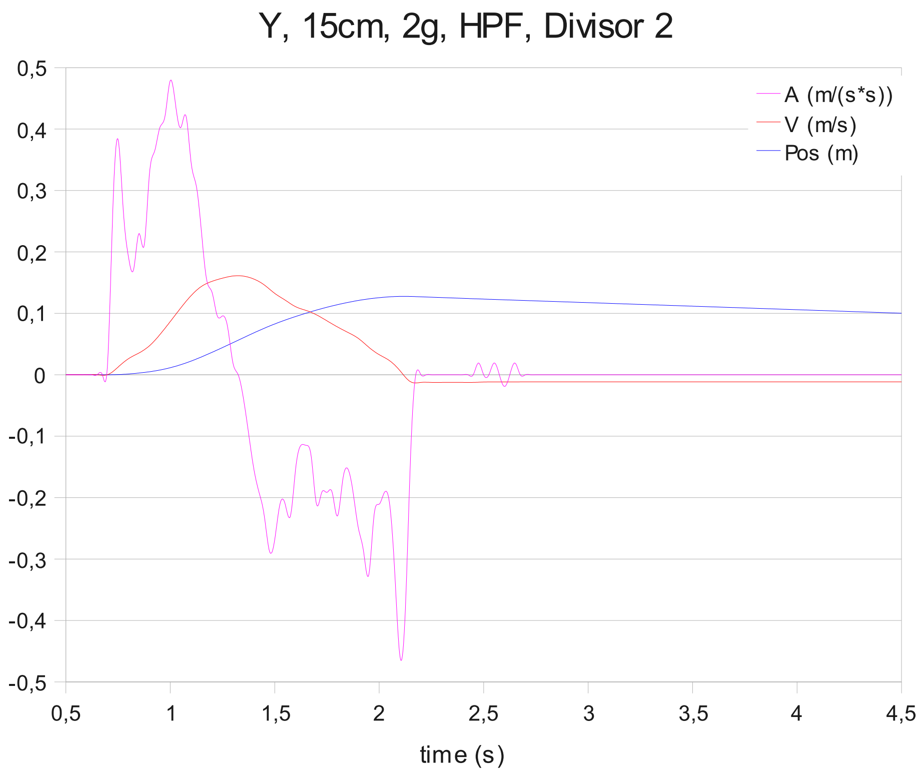

Figure 4.7: Acceleration, speed and position from figure 4.6 divided by 2.

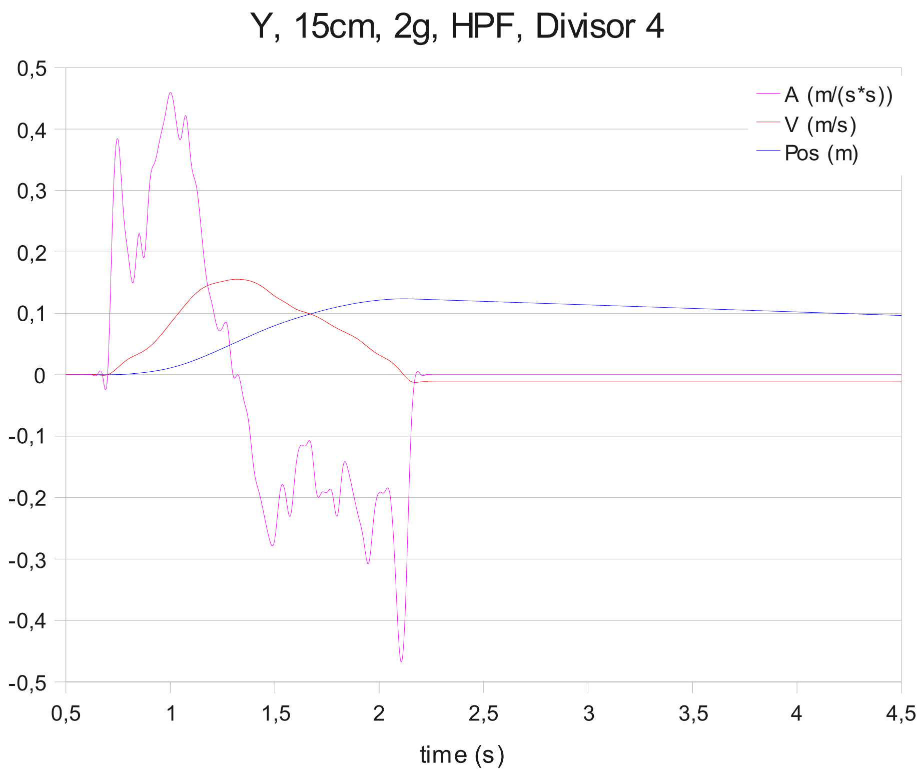

Figure 4.8: Acceleration, speed and position from figure 4.6 divided by 4.

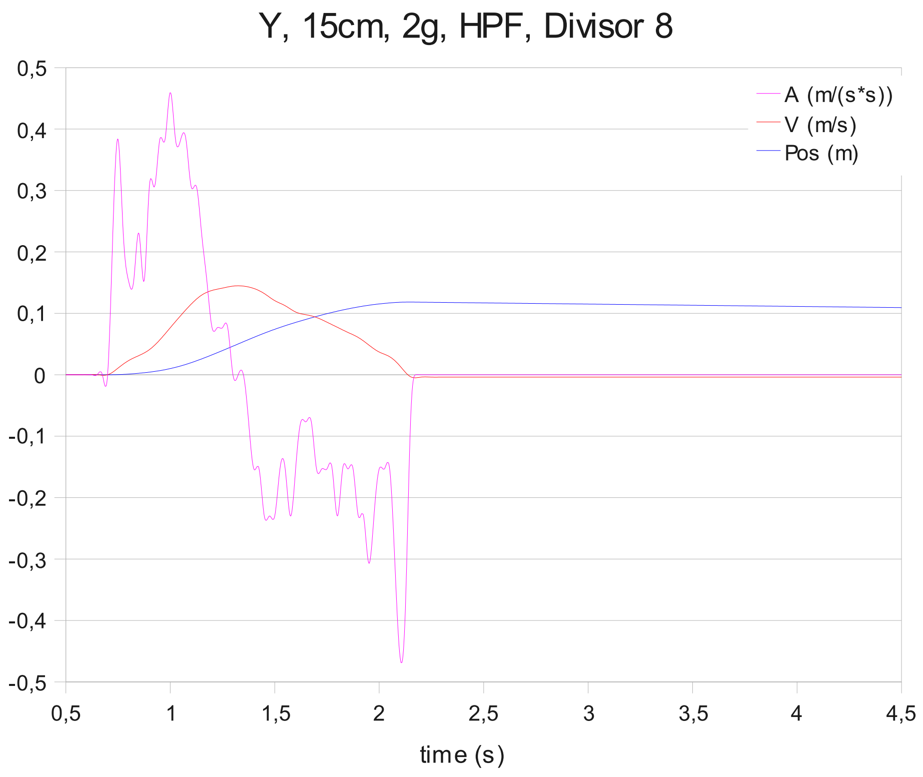

Figure 4.9: Acceleration, speed and position from figure 4.6 divided by 8.