UPGRADE OF A SCARA ROBOT USING OROCOS

Dalton Matsuo Tavares
Mechatronics Laboratory
University of São Paulo
Av. Trabalhador São-Carlense, 400 CEP 13566-590
São Carlos, São Paulo, Brazil
email: dmatsuo@sc.usp.br

Rafael Vidal Aroca and Glauco Augusto de Paula Caurin
Mechatronics Laboratory
University of São Paulo
Av. Trabalhador São-Carlense, 400 CEP 13566-590
São Carlos, São Paulo, Brazil
email: rafaelaroca@uol.com.br, gcaurin@sc.usp.br

ABSTRACT
This paper presents a bottom-up approach that permits the integration of new devices and functionalities into a robotic cell. Although there are currently notable efforts from the scientific community toward this goal, the initiative presented here combines some trends in architectural design from academic research institutes and tools produced by the free software community in order to seek a simple and accessible way to develop a modular environment for robot control. To demonstrate the features of this approach, this paper presents some results inspired by industrial and academic research, which illustrate an increasing degree of openness regarding robotic control. This is accomplished by the use of Linux Kernel Modules (LKMs) which insert new functionalities as they are needed, a real time operating system and the Open Robot Control Software project (Orocos).

KEY WORDS
Robot design and architecture, Free software, Orocos, Robot upgrade, Real time systems, RTAI.

1 Introduction

Due to demands in robustness and reliability it is not straightforward for researchers how to make improvements and contributions to industrial robotics. This happens due to the hardware and software architecture in use today which, in most cases, are partially or completely closed. This forces researchers to reverse engineer or retrofit industrial systems allowing them to access and control the servo control loop and other robot's subsystems.

Regarding Small and Medium Enterprises (SMEs) as well as research environments, where work is typically done with small to medium lot sizes, fast adaptability of the robot and the surrounding cell to new products and processes is much more important than keeping the cycle times small. To make this possible, the programming of robots (and especially robot cells) and the integration of new devices into these robot cells has to be much easier than today [1].

Considering this scenario, this paper presents a bottom up approach to implement an open robotic control system based on an upgraded IBM 7545 SCARA robot [2]. Similar to the approach presented at [3], the power drives were kept, but the original implementation regarding electronics and software were completely upgraded. The new system is entirely based on free software using a well-known real time Linux kernel extension which integrates new functions as kernel modules, granting the possibility of new devices integration. The main contribution of this approach is the support for the insertion of new functionalities as they are made available, with minimum disturbance to the operation of the system. The user space interaction counterpart was implemented using the Orocos robot control software framework[1].

2 Degrees of Openness

During the last decade, the openness of control systems has been addressed in several ways by worldwide research projects both in the field of machine tools (i.e. OSEC/JOP, OMAC, OSACA [4]) and in the field of Robotics (i.e. OROCOS [5]). Although an universal agreement on the definition of Open Architecture Control (OAC) has not yet been achieved, a basic set of common requirements can be drawn and summarized as follows [6]:

- an OAC allows the portability of software for different hardware platforms and operating systems;
- an OAC is modular having the capability of easy modules replacement with new ones, allowing scalability both in performance and functionality;
- an OAC allows easy addition of new functionalities. It is based on standards to let third parties develop hardware and software that meet new requirements and also provides plug-and-play mechanisms for fast integration of such new functionalities;
- an OAC is reconfigurable providing mechanisms for easy adaptation of its parameters in order to customize the system to different application scenarios;
- an OAC is economic since it is based on off-the-shelf modules both with reference to hardware and software.

[1] Available at: http://www.orocos.org/

Another key point of modern OAC is the use of free software in the form of open source software (OSS), which has attracted also the field of numerical controls for machine tools [7]. One of the most promising uses of OSS in OAC is at operating system level, which has the advantage of full source code availability that allows developers to provide their own customization; and additionally, allows the cost reduction of the whole controller cutting the license fee of commercial Real Time Operating Systems (RTOS) [6].

2.1 OAC Platforms

There exists today some architectures and systems characterized by several levels of openness. This section is not intended as a full survey on OACs, but it will present some initiatives (with an increasing level of openness) in the field of robotics. The approaches gathered here vary from manufacturer centered to open source centered. Manufacturer centered approaches are closed for the most part, although they allow some degree of freedom concerning the integration of new features. In this case the main point is to mask all the system complexities from the user in other to guarantee the ease of use and block the possibility of disturbance of the safety routines of the system by the user. Open source centered approaches focus on the possibility of including new features using a modular approach. In this case, a full knowledge of the system is demanded. Considering the roboticist as a developer, the last approach represents the trend of today's research institutes.

2.1.1 KUKA's Robot Sensor Interface

KUKA's Robot Sensor Interface (RSI) [8] is an example of quasi open standard robot system which uses a modular approach to integrate sensors into a control system. Its "openness" resides in the inclusion of predefined higher level system calls using a standard object library. Unlike the usual case, where the sensor system is connected by means of external interfaces, here the sensor is integrated using KUKA Robot Language (KRL). The sensor application itself is implemented in it. The "quasi open" part concerns the maintenance of a "developer level of protection". Although RSI allows a moderate degree of freedom considering the integration of supported sensors, the sensor drivers are locked from the user, as a mechanism to prevent users from disturbing the safety measures implemented. In a sense, it protects the user from itself. Although the available literature implies the developer is able to modify or implement new sensor drivers, it was not possible to find any concise information regarding this matter by the date of this submission.

2.1.2 Open Robot Control Architecture and Autonomous Reactive Robot Planning

Considering a higher level of openness, the Open Robot Control architecture (ORC) [3], created at the Lund Institute of Technology, developed an hierarchical architecture that integrates on-line and off-line programming at a higher level, creating specialized user views. At lower levels, it kept the original power drives and safety functions of the prototype robot[2] so that it could be more easily ported to an industrial environment. The infrastructure for sensor integration was developed considering this architecture as the basis.

The Autonomous Reactive Robot Planning (ARS) [9], developed at the Department of Mechanical Engineering at Lund University, proposed an approach for on-line integration between a simulated work cell (including simulated sensors) and the real work cell, allowing the retrofit of read data to the simulated model. Therefore, it is possible to insert new parameters that weren't foreseen in the off-line programming allowing the adaptation of the real work cell even after the upload of the control program.

2.1.3 Generator of Modules - GenoM

Generator of Modules (GenoM) [10] is an environment for description and implementation of robotic software components that provides the following:

- Component Model: GenoM defines specific interaction between components and composition standards;
- Component Model Implementation: GenoM provides the dedicated set of executable software elements required to support the execution of software components that conform to the model;
- Component Architecture: GenoM defines the internal architecture of software components, and their structure and functioning;
- Component Generation Tools: GenoM provides a set of tools for describing software components and for generating templates.

The generation process is based on a component description file that is parsed by the tool. Then, templates are generated and can be used by specialized roboticists for algorithms implementation. Finally, binary components are generated, in both real-time and non real-time forms, along with test programs.

2.1.4 Orocos Project

The Orocos project started as a free software project for robotics [11], but it has outgrown its robotics-dependent roots. "Orocos" is the acronym of the Open Robot Control

[2] An ASEA ABB Irb-2000/S3b system.

Software project. The project's aim is to develop a general-purpose, free software, and modular framework for robot and machine control. The Orocos project supports four C++ libraries: the Real-Time Toolkit, the Kinematics and Dynamics Library, the Bayesian Filtering Library and the Orocos Component Library[5].

The Orocos Real-Time Toolkit (RTT) is not an application in itself, but it provides the infrastructure and the functionalities to build robotics applications in C++. The emphasis is on real time, on-line interactive and component based applications. The Orocos Components Library (OCL) provides some ready to use control components. Both Component management and Components for control and hardware access are available. The Orocos Kinematics and Dynamics Library (KDL) is a C++ library which allows the calculation of kinematic chains in real time. The Orocos Bayesian Filtering Library (BFL) provides an application independent framework for inference in Dynamic Bayesian Networks.

3 Application Scenario

The development was based on a host-target scheme, where the software was developed on a Linux workstation and transparently sent to the target machine running a Real Time Linux (RTL) based on RTAI. It is clear that RTL, specifically RTAI, is more than adequate to control a robot with 6 degrees of freedom [12] with very low scheduling jitter (less than 10 μs in the worst case). Using a comparative study made by Aarno [13], it was possible to conclude that RTAI should preferably be used instead of other RTLs due to its better performance on stress.

An industrial PC was used for the real time system, consisting of a CompactPCI rack, with the following boards:

- Inova AMD K6-2 266 MHz CPU Board;
- Acromag Carriers: The CompactPCI carriers are boards that act as a bridge between the Industry Packs (IPs) and the CompactPCI Bus on the PC. Two carriers were used with two slots each;
- National Instruments I/O Board: CompactPCI board with 48 digital and analog inputs was adopted to read data provided by the six axis force/torque sensors placed at the robot wrist.

On the industrial PC world, the carrier boards can hold several smaller boards, each with a specific function. These boards are called IPs or Industry Packs. The used industry Packs were:

- Tews 48 Digital I/O IP: Industry Pack with digital inputs and outputs to read data from digital sensors and output digital data;
- Tews Quadrature IP: Industry Pack to read encoders values, and inform the software where the robot's axis are;
- Tews DAC IP: Industry Pack to convert digital data into analog signals, providing information to control the motors.

Other items used in the project were:

- Power Interface: Power boards to interface the digital I/Os from the IPs with the robot motors;
- Robot: An upgraded IBM 7545 SCARA robot.

Software drivers were developed to activate the robot motors, read the encoders and sensors, besides adaptations for operation with the industrial PC. Sections 3.1 and 3.2 will discuss the approach considering the software architecture.

3.1 Kernel Drivers

Before discussing the approach used to create a modular robot using real time OSS and Linux Kernel Modules (LKMs), it is necessary to define some terms. Since the operating system provides an hardware abstraction layer, and since it must provide resource sharing, every access to the hardware from user programs should be done through the operating system. To enforce this chain of access (in order to prevent malicious or poorly coded applications to damage the hardware) a protection is needed at the hardware level [14].

The hardware must provide at least two execution levels:

- Kernel mode: in this mode, the software has access to all the instructions and every piece of hardware.
- User mode: in this mode, the software is restricted and cannot execute some instructions, and is denied access to some hardware (like some area of the main memory, or direct access to the IDE bus).

So, at software level, two spaces are defined:

- Kernel space: code running in the kernel mode is said to be inside the kernel space.
- User space: every other program, running in user mode, is said to be in user space.

The decision to use LKMs is related to the reliability of kernel space when dealing with real time tasks provided by the RTAI kernel extension.

3.1.1 Background: Kernel Module Communication

Linux Kernel provides a simple and practical way to make one module communicate with each other, allowing seamless inter-module communication. Every written module can "export" one or more functions to other kernel modules, simply adding the macro $EXPORT\_SYMBOL$ at the end of the source code.

When a module exports a function that is used by other modules, a module dependency is automatically created, blocking users the ability to remove a module that is in use by another. In this situation, first the module that uses resources from a providing module should be removed, and in a second step, the module that provides the function could be removed. In a Linux Operating System, a list of kernel symbols, including the exported ones that could be used from any kernel module, can be obtained running the command '$cat/proc/ksyms$'.

3.1.2 LKM device drivers and real time access

On LKM programs, all real time tasks are implemented as running processes inside the Linux kernel as modules. The real time modules can be easily started or stopped with simple commands such as '$insmod$', that installs and starts running a new module, '$rmmod$', which removes and stops running a module from kernel space, and '$lsmod$' that can list all modules currently running. The most interesting aspect of this approach is that the system can admit modifications in its behavior, represented by the insertion of new modules, at run time, without taking it off-line.

The development of the low level drivers to access the hardware started at [2]. The carrier board device driver ($apc8620$) and the hardware abstraction layer (HAL) for the IPs ($carrier1\_hal$ and $carrier2\_hal$) were developed as a proof of concept. $carrier1\_hal$ and $carrier2\_hal$ were monolithic LKMs which incorporated the drivers for the, Quadrature IP, DAC IP ($carrier1\_hal$) and 48 Digital I/O IP ($carrier2\_hal$). In our first implementation, the user/kernel space communication was accomplished using a resource called real time FIFOs, which make FIFOs between the 2 operating system modes. Therefore, the FIFO should be implemented in both kernel and user space. A $scara\_fifo$ module was implemented to receive commands from user space programs, which sent data entered by users to the real time processes (e.g. a trigger to start an encoder reading, a discrete voltage to start motor movement, etc).

To prove the scalability of the LKM concept it was developed a LKM to control the bumper readings which are mapped on the 48 Digital I/O IP and a PID controller for each joint. Although the time constraints were met (a 5 μs resolution was reached for the PID and a 20 μs resolution for the bumper monitor) there was another problem: how to create an easy and standardized way for the user to interface with the robot environment? At this point the system required that the programmer knew the kernel implementation in order to develop user space programs to interact with the robot.

The most obvious answer to deal with this is to create a standard library with primitive functions to access the IPs and common tasks (e.g. homing, various controllers for the joints including PIDs, neural nets, etc.). The main objective now is to make the system accessible to the average user. Considering the approaches presented in Section 2, the most suitable to implement the user library is the one implementing a standard open framework. GenoM and Orocos are the obvious choices in this category. Orocos was chosen due to its free software nature, its integration with the RTAI kernel extension (which was already in use) and its extensive documentation which makes the access to its features a little bit easier.

According to the Orocos philosophy, it was necessary to make some minor adaptations on the original system. The first one was to modularize the implementation of the IP HAL LKMs. In the original version, the mere swapping of IPs among carrier slots would represent a major problem. This occurred because the IP device drivers were hard coded on the carrier initialization routines. The new version implements a separate module for the carrier, which initialize its two slots and a single module for each IP. This way, it is possible to swap IPs between carrier slots without any source code modification. The second (major) modification concerns the way user space programs access kernel space functions. Orocos uses LinuX RealTime (LXRT) to interface user space applications and low level kernel space device drivers. Therefore, the original implementation using real time FIFOs had to be adapted to cope with this prerequisite.

Although LXRT incurs in additional schedular latency, for most applications this slightly extra overhead worth the benefits. The major advantage is the possibility of development in soft real time, in which a buggy implementation won't take down the whole system, allowing the termination of any wild running process without a reboot. In [15], the measurements taken on a modest hardware (Intel PII 300Mhz) showed an additional 3μs overhead on top of the standard scheduler latency of 3μs.

The LXRT user/kernel space mapping is accomplished by means of two libraries: $lxrt\_user.h$ and $lxrt\_kernel.c$. $lxrt\_user.h$ creates an abstraction to the low level drivers so the kernel primitive functions can be accessed as a standard API (more about this on Section 3.2). $lxrt\_kernel.c$ is the LKM implementation which interfaces with the kernel space functions and exchanges information with them, intermediating requests for the user space library $lxrt\_user.h$.

3.2 User Space Library

The library responsible to link user space programs with kernel space functions is $lxrt\_user.h$. A sample entry for this file can be seen at Listing 1.

The main parameters on this code snippet are the $IP\_DIGITAL\_READ$ that is an index which uniquely identifies the kernel function (previously defined on $lxrt\_kernel.c$), the $\&arg$ which represents the address of the argument structure (in this case a dummy value considering this function does not have a parameter passed) and the $i[LOW]$ that is a return type of $rtai\_lxrt$ (a RTAI kernel module which makes LXRT API calls available) contains the return value of the kernel function.

Listing 1. $lxrt\_user.h$ snippet with the API mappings for the kernel space functions.

```
(...)
static inline int IP_Digital_read ()
{
  int retval;
  struct {unsigned int dummy; } arg = { 0 };
  retval = rtai_lxrt(MYIDX, SIZARG,
  IP_DIGITAL_READ , &arg).i[LOW];
  return retval;
}
(...)
```

Considering the other primitive methods are mapped in an analog fashion, the infrastructure for the implementation of the user space API primitives to access the robot hardware is complete. A simple library to execute the robot homing was created as a demonstration for some of the Orocos principles and functionalities.

The homing task uses Orocos libraries to define a task and its facilities; in this case methods[3] and events[4]. The Orocos template library $HomingInterface.hpp$ is also used to keep the homing task compliant with the Orocos methodology. The main functionalities available execute the homing procedure, generate a "home switch" event (which indicates the system is homed), return the homing status or reset it when desired. All of this is implemented using a C++ style language inherent to the framework.

Another interesting feature (for future works) is the possibility to define an Extensible Markup Language (XML) like file containing the initialization parameters for the task. So, the initial state of the task can be simply defined by modifying the parameters on this file.

Considering the additional facility of creating a finite state machine to represent the behavior of the system, all the basis to build the complete API library and programs for the upgraded IBM 7545 Scara robot is in place.

4 Current Limitations

Although it was possible to attain enough results to prove Orocos as a feasible approach for future developments at our laboratory, some design limitations are faced concerning the underlying real time system. There are some issues with RTAI which are not very well documented and we are not certain of its actual impact to the real time system as of now.

The load average[5] using the LXRT kernel module is too high (with values ranging from five to seven, five being the inherent load of the kernel module and the additional load being a characteristic of the real time process). This is a well known fact but relatively unexplored in the RTAI community. The only piece of information found concerning this matter explains this processes are in $UNINTERRUPTIBLE\_SLEEP$ state (characteristic of I/O processes), triggered by some internal daemons started by LXRT ($kthread\_b$, $kthread\_m$[6]), which are waiting for processes requests in $TASK\_UNINTERRUPTIBLE$ state. As a consequence, their cumulated wait time is charged as being actual load by Linux[7]. In our particular case, the process tree shows $kthread\_m$ starting four processes in the $UNINTERRUPTIBLE\_SLEEP$ state (Listing 2).

Listing 2. Process tree snippet generated by $pstree$.

```
init-+-RTAI_KTHRD_M:0-+-F:HARD:0:1
     |                |-F:HARD:0:2
     |                |-F:HARD:0:3
     |                `-F:HARD:0:4
(...)
```

Using the ps command, it is possible to see five $insmod\ rtai\_lxrt.ko$ processes in the $UNINTERRUPTIBLE\_SLEEP$ state. Considering the process tree and the sleep waiting processes listed at Listing 3, the relationship of a user real time process running in user space and its exact trajectory on the system internals is still a little bit fuzzy.

Listing 3. Process snippet generated by $psaux$.

```
(...) STAT START   TIME COMMAND
(...)
(...) D     12:34   0:00 insmod rtai_lxrt.ko
(...) D     12:34   0:00 insmod rtai_lxrt.ko
(...) D     12:34   0:00 insmod rtai_lxrt.ko
(...) D     12:34   0:00 insmod rtai_lxrt.ko
(...) D     12:34   0:00 insmod rtai_lxrt.ko
(...)
```

In addition to this, [16] describes LXRT intended purpose as an aid for an easier, less risky and faster development phase. If LXRT is intended as a tool for the development phase, what kind of reliability can be expected

[3] A task's methods are directly executed like a function and used to 'calculate' a result or change a parameter.

[4] A task's events are intended to be published by the task to subscribers, in order to 'notify' interested parties of a change in the system.

[5] Load average is an embedded metric that appears in the output of other UNIX commands like $uptime$ and $procinfo$. It's a time-dependent metric commonly used to observe system resource consumption. The load average tries to measure the number of active processes at any time. High load averages usually mean that the system is being used heavily and the response time is correspondingly slow. Ideally, a desirable load average value should be under three (...) Source: http://www.teamquest.com/resources/gunther/display/5/index.htm

[6] $kthread\_b$ is responsible for the first scheduling of any Linux schedulable object passing to work into hard real time. $kthread\_m$ is responsible for creating/deleting hard real time kernel threads, i.e what in kernel only schedulers is called a task. They take care of the threads reservoir also. Source: https://mail.gna.org/public/rtai-dev/2004-07/msg00067.html

[7] Source: https://mail.rtai.org/pipermail/rtai/2004-July/008077.html and https://mail.rtai.org/pipermail/rtai/2003-August/004634.html

on a system implemented on top of it? We still need to create stress tests so the system reliability can be verified. Besides, we need to test other options with Orocos. For instance, Orocos also supports Xenomai framework. Thus, future work involves comparisons between RTAI and Xenomai and a deeper analysis of the real time system internals[8].

5 Conclusion

Although the learning curve for the current control system was quite steep (2 months to master the framework and the necessary programming skills), the benefits obtained more than worth it. Different from other initiatives, which can be classified as a user mode framework with all the low level functions supplied by the manufacturer and shut down for the developer, Orocos allows the control of a robotic hardware in all its levels: user and kernel space. So, considering the correct hardware mappings for a given hardware are available (the Board Support Packages), the mapping of its functions and the insertion of additional hardware (e.g. new/legacy sensors) is feasible. That way the system can be perceived as a truly modular environment with extensions and development possible in all its levels (which is vital considering a mechatronics research group). The better part is the degree of openness of this approach. All the necessary tools, including the operating system kernel, the real time kernel extension and the user space framework are available as free software and fully compatible with each other.

6 Acknowledgements

This work was developed in the Mechatronics Laboratory at University of São Paulo (USP). Thanks to Leonardo Marquez Pedro, who helped with the mechanics of the SCARA Robot, Sérgio for the electronics support and Frederic Pont, Michael Gauss and Rafael Aroca for the enlightenment on robotics and real time. We also thank FAPESP[9] for the financial support.

7 References

[1] M. Naumann, K. Wegener, R. Schraft, and L. Lachello, Robot cell integration by means of application-p'n'p, *In VDI-Wissensforum et al.: ISR 2006 – ROBOTIK 2006 – Proceedings of the Joint Conference on Robotics*, 2006.

[2] R. V. Aroca, D. M. Tavares, G. Caurin, Scara Robot Controller Using Real Time Linux, *2007 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, September, 2007.

[3] K. Nilsson, Industrial robot programming, *Ph.D. dissertation*, Department of Automatic Control – Lund University, 1996.

[4] O. Asato, E. Kato, and R. I. et al., Analysis of open cnc architecture for machine tools, *Journal of the Brazilian Society of Mechanical Sciences*, 2002, [On-line]. Available: http://www.scielo.br/scielo.php?script=sci arttext&pid=S0100-73862002000300009&lng=en&nrm=iso. ISSN 0100-7386. doi:10.1590/S0100-73862002000300009.

[5] H. Bruynickx and P. Soetens, The OROCOS Project, 2007, [On-line]. Available: http://www.orocos.org/orocos/whatis

[6] D. Colombo, D. Dallefrate, and L. M. Tosatti, Pc based control systems for compliance control and intuitive programming of industrial robots, *In VDI-Wissensforum et al.: ISR 2006 – ROBOTIK 2006 – Proceedings of the Joint Conference on Robotics*, 2006.

[7] M. de Sousa, Matplc-the truly open automation controller, *In IECON 02 – IEEE 2002 28th Annual Conference of the Industrial Electronics Society*, 2002, 2278–2283, ISBN: 0-7803-7474-6.

[8] K. R. GMBH., KR C1, KR C2 – Robot Sensor Interface (RSI) - Release 1.0 (BETA), 2006.

[9] M. Olsson, Simulation and execution of autonomous robot systems, *Ph.D. dissertation*, Department of Mechanical Engineering, Lund University, 2002.

[10] F. Pont, R. Siegwart, A Real-Time Software Framework for Indoor Navigation. *In Proceedings of the IEEE/RSJ Intenational Conference on Intelligent Robots and Systems*, IROS'2005, Edmonton, Canada.

[11] H. Bruynickx, Open robot control software: the orocos project. *In IEEE International Conference on Robotics and Automation*, 2001, 2523–2528.

[12] D. Aarno[a], Control of a puma 560 using linux real-time application interface (rtai), 2004 [On-line]. Available: http://www.nada.kth.se/bishop/resources/rtcontrol.pdf

[13] D. Aarno[b], Evaluation of real-time linux derivatives for use in robotic control, 2004 [On-line]. Available:http://www.nada.kth.se/ bishop/resources/rtos.pdf.

[14] G. L. Mignot, Monolithic kernels and micro-kernels, June, 2005 [On-line]. Available: http://kilobug.free.fr/hurd/presen/abstract/html/node2.html

[15] H. Kuppens, Realtime Linux (RTAI) - Radboud University Experiment #3 LinuX RealTime - LXRT, 2006 [On-line]. Available: http://www.cs.ru.nl/lab/rtai/experiments/3.LXRT/Experiment-3.html

[16] P. Mantegazza, DIAPM RTAI for Linux: WHYs, WHATs and HOWs, *Real Time Linux Workshop*, Vienna University of Technology, Dec. 1999, [On-line]. Available: https://www.rtai.org/index.php?module=documents &JAS_DocumentManager_op=downloadFile&JAS_File_id=31

[8] Probably using VxWorks Scope Tools.

[9] Fundação de Amparo a Pesquisa do Estado de São Paulo, process number 2004/14648-3