DEPARTMENT OF ELECTRONIC & ELECTRICAL ENGINEERING

University of Strathclyde Engineering

ITRA - Reference manual

VERSION 2.0 - February 2019

Carmelo Mineo[1,*], Cuebong Wong[2], Momchil Vasilev[1], Bruce Cowan[1], Charles N. MacLeod[1], S. Gareth Pierce[1], Erfu Yang[2]

[1] Department of Electronic and Electrical Engineering, University of Strathclyde, Royal College Building, 204 George Street, Glasgow G1 1XW, UK

[2] Design Manufacture & Engineering Management, University of Strathclyde, James Weir Building, 75 Montrose Street, Glasgow G1 1XW, UK

* *carmelo.mineo@strath.ac.uk*

Contents

Overview 3
Reference 3
Requirements 4
Hardware requirements 4
Safety 4
Architecture 5
ITRA Functions 8
Initializers 8
Networking 8
Getters 8
Setters 9
Other function classifications 10
Classification according to the call mode (before, after or before/after) 10
Classification according to the output type 10
Classification according to the task 11
Detailed description of function usage 12
External control application examples 17
Introduction 17
Preliminary installation steps 17
KRL-based approach 18
Computer-based approach 19
RSI-based approach 20
Benchmarking 22
Function run-times 22
External control reaction times 22

Overview

Robots are increasingly present in industry. Achieving effective integration and the full potential of robotic systems presents significant challenges. Robots, sensors and end-effector tools are often not necessarily designed to be put together and form a system. This manual introduces a C++ language-based toolbox, designed to facilitate the integration of industrial robotic arms with server computers, sensors and actuators. The toolbox, named as Interfacing Toolbox for Robots Arms (ITRA), contains fundamental functionalities for robust connectivity, real-time control in Cartesian and joint space and auxiliary functions to set or get key functional variables. It is designed to run on a remote computer connected with one or multiple robot controllers. All embedded functions can be used through high-level programming language platforms (e.g. MATLAB®, LabVIEW®), providing the opportunity to speed-up robust integration of robotic systems. Emerging applications aim to use robot arms in changing environments with movable obstacles or where the shape of the surroundings is changing. In such situations, the robots need to adapt their tasks/behaviours. ITRA is a C++ based library with functions using C calling conventions. ITRA makes use of standard C++14 and Boost, and is therefore cross-platform, and can be compiled as a dynamic link library (DLL) for Windows, and as a shared object (SO) for Linux-based operating systems. ITRA contains functions designed to enable real-time adaptive robot behaviour, maximizing the robot promptness and respecting constraints (maximum accelerations and velocities). The toolbox is compatible with all KUKA robotic arms, based on the fourth generation of KUKA controllers and equipped with the Robot Sensor Interface (RSI) software add-on. The current version of ITRA is available for Windows 64bit platforms and Linux platforms.

The ITRA DLL for Windows based computers is constituted by the following files:

robotComms.dll – the DLL binary file

robotComms.h – the header file with the list of exposed functions

The ITRA SO for Linux environments is constituted by:

librobotComms.so – the SO binary file

robotComms.h – the header file with the list of exposed functions

Reference

C. Mineo, C. Wong, M. Vasilev, C. N. MacLeod, S. G. Pierce and E. Yang, *Software Interfacing Toolbox for Robotic Arms with Real-Time Adaptive Behavior Capabilities*, to be submitted to IEEE Journal, 2018.

Requirements

Before starting to use the Toolbox for controlling the robot, the following software/hardware are required:

Hardware requirements

The user is required to have:

1. 6-DoF KUKA robot(s) based on KRC4 controllers;
2. An up-to-date external PC/laptop with good computational power.
    a. Processor-supported external system with real-time-capable network card with 100 Mbit in full duplex mode.
3. Network cable for switch, hub or crossed network cable for direct connection
    a. Good Ethernet cable (category five or better).

A network between the robot and the PC shall be established. To establish the network, the user shall do the following:

- Using an Ethernet cable, connect the X66 port of the robot to the Ethernet port of your PC.
- From the teach pendant of the manipulator, verify the IP of the robot.
- On the external PC, the user shall change the IP of the PC into a static IP in the range of robot's IP.

Software requirements

The following software packages are required:

- KUKA RobotSensorInterface (RSI) 3.1 or higher.
- KUKA System Software 8.3.
- Computer with Windows 64bit operating system or Linux operating system.
- On the RSI XML file loaded into the robot controller, to support the required RSI-Visual configuration, the same IP address used by the external computer must be specified.

Safety

This documentation contains safety instructions which refer specifically to the ITRA software toolbox described here. For fundamental safety information of the industrial robot(s) in use, the users are directed to the "Safety" chapter of the KUKA robot reference manuals and the "safety" chapter of the KUKA RSI reference manual [1].

WARNING FOR USING ITRA FOR ANY FORM OF REAL-TIME ROBOT CONTROL

Incorrect use of KUKA RobotSensorInterface and of the ITRA toolbox can cause personal injury and material damage. The authors of ITRA cannot be considered responsible for any form of damage.

In real-time external control operation, the robot may move unexpectedly in the following cases:

- Incorrectly parameterized RSI objects;
- Incorrectly parameterized ITRA functions;
- Hardware fault (e.g. incorrect cabling, break in the sensor cable or malfunction sensors providing wrong data);

Unexpected movements may cause serious injuries and substantial material damage. The system integrator is obliged to minimize the risk of injury to himself/herself and other people, as well as the risk of material damage, by adopting suitable safety measures, e.g. by means of workspace limitation and barriers.

Architecture

The fourth generation of KUKA robot controller (KRC4) is divided into three main systems, as it is shown in Fig. 1. The graphic user interface (GUI) allows the user to write and execute robot programs, through defining robot bases, tool parameters and by jogging the robot arm. This GUI runs within an embedded version of Windows XP®. Hidden from the user is a separate operating system called VxWorks®. This is a real-time operating system, which is designed for embedded applications and is developed by Wind River Systems [2]. The VxWorks system controls all robot drives and is used because of its multi-tasking capabilities, real-time performance and reliability.

Although running on the same processor, the Windows XP and VxWorks operating systems are entirely separate from each other. Any information that is passed between them is sent over a virtual TCP/IP connection within the KRC architecture. There is no physical network cable but information is packed up, transmitted over the virtual connection, received and unpacked by the other system to be processed.

ITRA is compatible with all KRC4 robots equipped with a KUKA software add-on known as Robot Sensor Interface (RSI) [1]. RSI runs under the VxWorks operating system in a real-time manner. It was purposely developed by KUKA to enable the communication between the robot controller and an external system (e.g. a sensor system or a server computer). Cyclical data transmission from the robot controller to the external system (and vice-versa) takes place in parallel to the execution of the KUKA Robot Language (KRL) program. Using RSI makes it possible to influence the robot motion or the execution of the KRL program by processing external data. The robot controller communicates with the external system via the Ethernet UDP/IP protocol. No fixed data frame is specified. The user must configure the template of the structure and the content of the data packets in an XML file, stored in the robot controller. Typical data packets, sent as ASCII packets by RSI to the external system, can include feedback Cartesian or axial coordinates, status of digital I/O signals and real-time operating parameters (e.g. drives currents and torques).

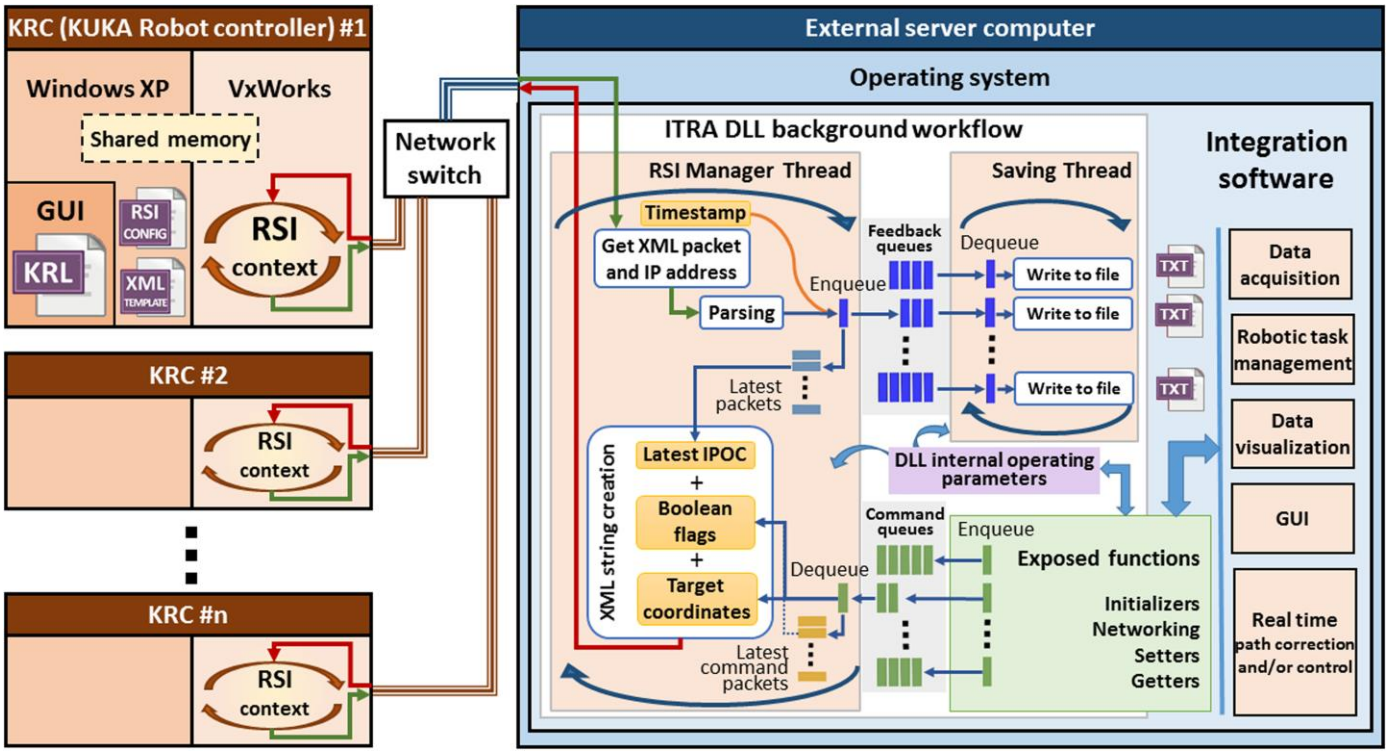

Fig. 1. Schematic representation of the architecture of the KRC4 controller and of the ITRA toolbox.

Typical data packets received from the external system can include a number of Boolean, integer or double precision variables. Fig. 2 shows the XML template file defining the content of the packets transferred between RSI and the server computer, supporting all functionalities of the ITRA toolbox. The first part of the file

comprises the connection parameters. IP_NUMBER and PORT are respectively the IP address and the port of the external system UDP socket. SENTYPE is the identifier of the external system; it is checked by RSI to validate every data packet it receives. ONLYSEND defines the direction of the data exchange; FALSE indicates that RSI sends and receives data. The signals from the RSI context that are sent to the external system are defined in the SEND section. From this XML section, RSI automatically creates the XML ASCII packet that the KRC transmits. It includes the Cartesian actual coordinates (incorporated through the "DEF_RIst" keyword), the Axis-specific actual position of robot axes A1 to A6 (incorporated through the "DEF_AIPos" keyword) and the status of four KRC digital outputs. The ASCII packet received from the external system is parsed by the RSI context accordingly to the XML template contained within the RECEIVE section. The RSI expects to receive eight double precision values and four Boolean values. The HOLDON attribute is set equal to "1", to make sure that, if a data packet arrives too late to the RSI context, the most recent valid value is maintained in place of the value expected from the external system.

The data packet received from the external system is processed within each machine cycle according to a data processing algorithm defined in the RSI configuration. That is generated through an object-based programming software application known as "RSI-Visual", using a library of RSI objects. Each RSI object performs a specific function with its signal inputs and makes the result available at its signal outputs. The linking of the signal inputs and outputs of multiple RSI objects creates a signal flow. The overall signal flow is called "RSI context". In the KRL program, the RSI context can be loaded and the signal processing parallel to program execution can be activated and deactivated. The signal processing is performed at the RSI cycle rate. Two cycle durations are available: 12 ms and 4 ms.

```
<ROOT>
  <CONFIG>
     <IP_NUMBER>10.1.1.1</IP_NUMBER>
     <PORT>49152</PORT>
     <SENTYPE>Server</SENTYPE>
     <ONLYSEND>FALSE</ONLYSEND>
  </CONFIG>

  <SEND>
     <ELEMENTS>
        <ELEMENT TAG="DEF_RIst"  TYPE="DOUBLE" INDX="INTERNAL"/>
        <ELEMENT TAG="DEF_AIPos" TYPE="DOUBLE" INDX="INTERNAL"/>
        <ELEMENT TAG="Digout.o1" TYPE="BOOL"   INDX="1"/>
        <ELEMENT TAG="Digout.o2" TYPE="BOOL"   INDX="2"/>
        <ELEMENT TAG="Digout.o3" TYPE="BOOL"   INDX="3"/>
        <ELEMENT TAG="Digout.o4" TYPE="BOOL"   INDX="4"/>
     </ELEMENTS>
  </SEND>

  <RECEIVE>
     <ELEMENTS>
        <ELEMENT TAG="D1" TYPE="DOUBLE" INDX="1"  HOLDON="1"/>
        <ELEMENT TAG="D2" TYPE="DOUBLE" INDX="2"  HOLDON="1"/>
        <ELEMENT TAG="D3" TYPE="DOUBLE" INDX="3"  HOLDON="1"/>
        <ELEMENT TAG="D4" TYPE="DOUBLE" INDX="4"  HOLDON="1"/>
        <ELEMENT TAG="D5" TYPE="DOUBLE" INDX="5"  HOLDON="1"/>
        <ELEMENT TAG="D6" TYPE="DOUBLE" INDX="6"  HOLDON="1"/>
        <ELEMENT TAG="D7" TYPE="DOUBLE" INDX="7"  HOLDON="1"/>
        <ELEMENT TAG="D8" TYPE="DOUBLE" INDX="8"  HOLDON="1"/>
        <ELEMENT TAG="B1" TYPE="BOOL"   INDX="9"  HOLDON="1"/>
        <ELEMENT TAG="B2" TYPE="BOOL"   INDX="10" HOLDON="1"/>
        <ELEMENT TAG="B3" TYPE="BOOL"   INDX="11" HOLDON="1"/>
        <ELEMENT TAG="B4" TYPE="BOOL"   INDX="12" HOLDON="1"/>
     </ELEMENTS>
  </RECEIVE>
</ROOT>
```

Fig. 2. RSI XML template supporting the functionalities of the ITRA toolbox.

When the RSI context is activated, external data are processed by RSI and forwarded to a portion of the KRC memory that can be accessed by the KRL program. Appended to the end of every packet sent by RSI is a number identified as the Interpolation Cycle Counter (IPOC), which indicates the current timestamp of the data packet. RSI expects the external system to extract this timestamp and append it to the return packet, which must be received by the RSI context within the same cycle. If RSI does not receive the IPOC number back within the cycle duration, the packet is deemed late [1].

ITRA is a C++ based library with functions using C calling conventions. ITRA makes use of standard C++14 and Boost, and is therefore cross-platform, and can be compiled as a dynamic link library (DLL) for Windows, and as a shared object (SO) for Linux-based operating systems. ITRA was designed to get feedback parameters from one or more robots simultaneously, to monitor the status of the running KRL robot programs and trigger the progress of the robotic tasks from a server computer. The C++ language was chosen to develop the library, since it is particularly suitable to develop highly robust communication and data processing algorithms that run in a reliable real-time manner. This language offers the programmer specific features to avoid the periodic, automated creation and disruption of allocated memory, known as garbage collection [3]. Other languages (e.g.

C#), which do not allow the same level of control on the allocated memory, can lead to unexpected drops in software performances [4, 5].

The ITRA architecture is described below. The reader can refer to the schematic representation given in Fig. 1. Once ITRA is loaded into a hosting programming environment (e.g. LabView or MATLAB), running within the operating system of the server computer, the constructor initializes fundamental variables to support UDP/IP connection with the robots. These are private variables that cannot be accessed by the hosting application. However, a certain level of control of the ITRA internal operating parameters is available through some of the public functions (described below), which allow specifying the number of robots to manage, their IP addresses and the directory that ITRA uses to store data. Only one socket is prepared by the constructor, to communicate with all robots. The connection socket is open through the "openConn" function (see below). At this stage, ITRA does not manage any data packets received from the robots. Since each RSI XML packet must get a reply packet from the external system, ITRA needs to run a background thread that receives the RSI packets, parses the data, extracts the packet IPOC numbers and mirrors them to the robots. Such thread is critically important to maintain a robust communication with the robots. It is hereafter referred as RSI-Manager Thread (RMT). RMT cyclically checks if data are available on the UDP socket. As soon as a XML packet is in the socket, the RMT takes a high resolution clock timestamp and downloads the packet from the socket, decoding the IP address of the KRC that sent it. The IP address is used to identify the index associated to the robot. Then, the XML packet is parsed to extract the Cartesian and axial coordinates, the status of the digital outputs and the packet IPOC number.

It may be necessary to store the parsed positional feedback. Since writing data to files can cause disrupting delays in the RMT, ITRA uses a secondary auxiliary thread, hereafter referred as Saving Thread (ST). The transfer of the parsed data packets takes place through FIFO queues. These are container adaptors specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other end [3]. The number of FIFO queues initialized by ITRA is equal to the number of connected robot controllers, so the data packets arriving from a robot controller are sent to the queue identified by the same robot index. Each data packet is enqueued jointly with the timestamp taken at the time of reception. The ST continuously looks for new packets in the queues and competes against the RMT to empty the containers. Since these queues are used to hold robot feedback data, they are referred as "feedback queues" in Fig. 1. Besides sending each received data packet and its timestamp to a queue, a copy of the timestamped data is temporarily stored into a structured array containing the latest packets received from each robot controller. Every time a new packet is received from the i-th robot, the i-th element of the array is refreshed with the new data. This is useful to keep a copy of the most recent data received from the robots, even when the feedback queues are completely emptied by the ST.

Although the ST is initialized when the RMT is launched, it does not save any data packet into file by default. This is to enable the user to specify when it is necessary to save the robot positional feedback. An ITRA function (see below) allows enabling/disabling the saving of the positional feedback for each robot, specifying the data format to be sent to file. The ST creates a separate text file (.txt) for each connected robot, appending the feedback positional packets to the end of the files, every time the saving is enabled.

The hosting application can use the public functions of ITRA. These functions support the development of simple and complex integration software platforms, comprising modules like data acquisition, multiple robot task synchronization, interfacing with sensors, data visualization, robot path control and graphical user interfaces.

ITRA Functions

ITRA contains 25 public functions, which can be divided in four groups, as it is shown in Table I. A general and detailed description of the functions is given below.

Initializers

The functions referred as "Initializers" are designed to set internal fundamental operating parameters of ITRA (e.g. number of robots, IP addresses, type of connection and output directory). These functions can only be used before launching the background service threads (RMT and ST), except for setRobFeedbackOutput. This function sets the format of the positional feedback to store into files. It can be called before launching the threads, to pre-set the behavior of the ST at the start, or during runtime to enable/disable the saving of the positional feedback for one or more robots.

Networking

The networking functions allow opening the UDP connection, checking if data are available in the socket, starting the RMT to manage the connection with the robots, terminating the background service threads when they are no longer required and closing the connection. The saving thread is automatically launched and terminated together with the RSI-manager thread. The terminating function always waits for the feedback queues to become empty before killing the threads, to avoid that some required positional feedback packets remain in the private queues and are never sent to file.

TABLE I
LIST OF ITRA FUNCTIONS DIVIDED INTO GROUPS

	Function names	Description
Initializers	setNumRob	Set number of robots to manage
	setRobIP	Set IP address of robot(s)
	setRobConnType	Set connection type (receive or receive/send)
	setOutputDir	Set directory for saving feedback file
	setRobFeedbackOutput	Set format of positional feedback to store
Networking	openConn	Open connection socket
	isDataAvailable	Check if data are available in the socket
	startRSIManager	Start RSI Manager Thread (RMT)
	terminateRSIManager	Terminate RMT
	closeConn	Close connection socket
Getters	isRSIRunning	Check if RSI is running on a specific robot
	isRobotTaskActive	Check if the robot task is active
	isRobStill	Check if the robot is still
	isRobMoveRequired	Check if a robot move is required
	isDataAcquRequired	Check if data acquisition is required
	getCurrPos	Get current robot position
	getTimestamp	Get current time
Setters	allowRobotStart	Allow robot to start its task
	allowRobMove	Allow robot to move
	allowRobotFinish	Allow robot to finish its task
	requRealTimeEnd	Request termination of real-time control
	requRobTaskEnd	Request termination of current robot task
	setCartPos	Set target position in Cartesian space
	setAxialPos	Set target position in joint space
	setToolPathFromFile	Set external control tool-path from file

Getters

The "Getters" are functions able to retrieve data required by the hosting application. They query the structured array containing the latest packets received from the robot controllers. The function to get the current robot position accesses the requested element of the array and retrieves the parsed Cartesian and axial coordinates, returning them to the hosting application as an array of double precision values. These can be used to monitor the robot position remotely from the server computer or to encode sensor data in a real-time fashion. Other

getters return a Boolean value (TRUE or FALSE); these ITRA functions operate based on the conventional meaning given to the status of the four digital outputs inserted by RSI into the XML packets (see details below). The function that gets the current clock time (the current timestamp) is the only function that does not queries the array with the latest packets. It retrieves the current value of the internal ITRA performance counter, which is a high resolution clock. The function returns a double precision value with the timestamp expressed in microseconds (μs). ITRA performance counter is the same clock used to timestamp the received packets sent to the feedback queue and (optionally) stored into files. Getting access to the same clock used to timestamp the feedback positional packets can be very useful, for example when it is necessary to encode sensor data through interpolated robot positions.

Setters

The "Setters" are functions able to influence the execution of predefined KRL programs and/or to control the robot tool-path. When called by the hosting applications, these functions generate command data packets addressed to one of the connected robots. The index of the target robot is given to the setters as an input. The generated command packets are sent to reserved FIFO queues, separated from the feedback queues. Such containers are referred as "command queues" (see Fig. 1) and they are also initialized by the ITRA constructor as soon as the library is loaded into the hosting application. The number of command queues is equal to the number of connected robot controllers, so each command packet can be sent to the queue identified by the same robot index targeted by the hosting application. The command packets are dequeued by the RSI-Manager Thread. After parsing the RSI packet received from the i-th robot controller, the RMT must reply to the robot through an XML string containing the data described in the RECEIVE section of the XML template (Fig. 2). The RMT looks for command packets available in the i-th command queue. If the queue is not empty, the packet at the front of the queue is dequeued and its content is concatenated into a string, according to the XML format expected by the RSI context. The setters allow flexible control of the robot arms, through the conventional meaning given to the value of the variables inserted into the XML packets sent to the robot controllers. Through some of the setters, the hosting application can trigger a robot to start its task, to continue the task (e.g. after a phase during which the robot must be still) or allow the robot to terminate the task and return to the home position. Such type of control is achieved through acting on the values of the four Boolean variables, denoted as B1-B4 in Fig. 2. These critically important logical setters use software handshaking to guarantee the robustness of the messaging between robot controllers and external computer; they expect to receive a change in the status of the digital flags sent by RSI (the four KRC digital outputs), as an acknowledgement for the successful communication. Permission to proceed the execution of the KRL program is not granted to the robot, if such acknowledgement if not received.

Sending target positions to the robot controllers, it is possible to control the robot tool-paths from the external computer. ITRA has functions to set command coordinates in Cartesian-space and in joint-space. External robot control is achieved by transmitting the command coordinates through six of the double precision variables (D1-D6). The preferred robot speed and acceleration can also be controlled through the two remaining variables (D7 and D8). Further details are given below. Each command packet dequeued from the i-th command queue is also used to refresh the i-th element of a structured array containing the latest command packets sent to each robot controller. The copy of the latest command position sent to the i-th robot is used when the external path-control is active and the i-th command queue does not contain any new command packets. This ensures the robot reaches the latest commanded position and stops there, until a new target position is requested by the external computer.

Other function classifications

Beside the grouping described above, it is possible to group the ITRA functions according to the possibility to call the function before the ITRA threads become alive or only after, according to the type of output, and according to the task.

Classification according to the call mode (before, after or before/after)

Before	After	Before/After
SetNumRob(numRobots)	isRSIRunning(robIndex)	getTimestamp()
SetOutputDir(dirString)	isRobotTaskActive(robIndex)	setRobFeedbackOutput(robIndex,val)
SetRobConnType(robIndex,conType)	isDataAcquRequired(robIndex)	
setRobIP(robIndex,robotIP)	isRobMoveRequired(robIndex)	
openConn(localIP,localPort)	isRobStill(robIndex)	
isDataAvailable()	terminateRSIManager()	
startRSIManager()	getCurrPos(robIndex, *feedbackPos)	
closeConn()	allowRobotStart(robIndex)	
	allowRobMove(robIndex)	
	allowRobotFinish(robIndex)	
	requRobTaskEnd(robIndex)	
	requRealTimeEnd(robIndex)	
	setCartPos(robIndex,x,y,z,a, b,c,speed,acc,mode)	
	setAxialPos(robIndex,a1,a2,a3, a4,a5,a6,speed,acc,mode)	
	setToolPathFromFile(robIndex, strPath)	

Classification according to the output type

void	bool	double
setRobFeedbackOutput(robIndex,val)	openConn(localIP,localPort)	getTimestamp()
getCurrPos(robIndex, *feedbackPos)	isDataAvailable()	
startRSIManager()	isRSIRunning(robIndex)	
SetNumRob(numRobots)	isRobotTaskActive(robIndex)	
SetOutputDir(dirString)	isDataAcquRequired(robIndex)	
SetRobConnType(robIndex,conType)	isRobMoveRequired(robIndex)	
setRobIP(robIndex,robotIP)	isRobStill(robIndex)	
terminateRSIManager()		
allowRobotStart(robIndex)		
allowRobMove(robIndex)		
allowRobotFinish(robIndex)		
requRobTaskEnd(robIndex)		
requRealTimeEnd(robIndex)		
setCartPos(robIndex,x,y,z,a,b,c,speed,acc,mode)		
setAxialPos(robIndex,a1,a2,a3,a4,a5,a6,speed,acc,mode)		
setToolPathFromFile(robIndex,strPath);		
closeConn()		

Classification according to the task

Managing the connection	Robot feedback/external control	Robot status	Timestamp
isDataAvailable()	setRobFeedbackOutput(robIndex,val)	isRobotTaskActive(i)	getTimestamp()
startRSIManager()	getCurrPos(robIndex, *feedbackPos)	isDataAcquRequired(i)	
isRSIRunning(robIndex)	SetOutputDir(dirString)	isRobMoveRequired(robIndex)	
openConn(localIP,localPort)	setCartPos(robIndex,x,y,z,a,b,c,speed,acc,mode)	isRobStill(robIndex)	
terminateRSIManager()	setAxialPos(robIndex,a1,a2,a3, a4,a5,a6,speed,acc,mode)		
SetRobConnType(robIndex,conType)	setToolPathFromFile(robIndex, strPath);		
setRobIP(robIndex,robotIP)	allowRobotStart(robIndex)		
SetNumRob(numRobots)	allowRobMove(robIndex)		
waitRSIReadiness(robIndex)	allowRobotFinish(robIndex)		
closeConn()	allowRobotStart(robIndex)		
	requRobTaskEnd(robIndex)		
	requRealTimeEnd(robIndex)		

Detailed description of function usage

```
void setNumRob(int numRobots);
```

The function sets the number of robots to manage through the "RSI-Manager Thread".

```
void setRobIP(int robIndex, string robotIP);
```

The function sets the IP address (robotIP) of the robot with index given by "robIndex".

```
void setOutputDir(string dirString);
```

The function sets the directory path (dirString) to use to save the robot feedback, through the library Saving Thread.

```
void setRobConnType(int robIndex, bool conType);
```

The function sets the type of connection to establish with the robot with index given by "robIndex".

conType == 1 => RSI-Manager Thread configured in "only receive" mode.

conType == 0 => RSI-Manager Thread configured in "receive and send" mode.

```
bool openConn(string localIP, int localPort);
```

The function opens the connection with the robots (one or more). *localIP* is the IP address of the local machine and *localPort* is the UDP port to use.

```
bool closeConn();
```

The function closes the connection with the robots (one or more).

```
bool isDataAvailable();
```

The function checks if there is data available in the UDP socket. It returns "true" if there are data available, "false" if the socket is empty. It can be used to postpone the start of the RSI-Manager Thread to the time when it becomes needed (there is no need to start the RSI-Manager Thread if none of the robots is sending packets).

```
void startRSIManager();
```

The function starts the RSI-Manager Thread and the Saving Thread. The Saving Thread gets alive; however, it does not send any timestamped packet to file by default.

```
void terminateRSIManager();
```

The function terminates the RSI-Manager Thread and the Saving Thread.

```
void setRobFeedbackOutput(int robIndex, int val);;
```

The function configures the Saving Thread for the robot with index "robIndex".

"val" must be between 0 and 3.

val == 0; nothing is sent to file.

Val == 1; Cartesian coordinates are sent to file.

Val == 2; Axial coordinates are sent to file.

Val == 3; Cartesian and axial coordinates are sent to file.

The following examples are useful to understand the function.

```
setRobFeedbackOutput(0,0); - no packets are sent to file for robot #1 (with index 0);

setRobFeedbackOutput(2,0); - no packets are sent to file for robot #3 (with index 2);

setRobFeedbackOutput(0,1); - Cartesian coordinates from Robot #1 are sent to file
                             (rob_1_Feedback.txt);

setRobFeedbackOutput(0,1); - Cartesian coordinates from Robot #1 and #2 are appended
setRobFeedbackOutput(1,1);   to file. (rob_1_Feedback.txt for Robot #1 packets and
                             rob_2_Feedback.txt for Robot #2 packets);

setRobFeedbackOutput(3,2); - Axial coordinates from Robot #4 are appended to file.
                             (rob_4_Feedback.txt);

setRobFeedbackOutput(2,3); - Cartesian and axial coordinates from Robot #3 are
                             appended to file (rob_3_Feedback.txt);
```

The function can be executed before starting the RMT, to pre-set the behaviour of the Saving thread, or during the execution of the RMT. Therefore, the function can be used to interrupt or enable the packet saving for periods of time for one or more robots. The files with the received and timestamped packets are stored within the same directory specified through the *SetOutputDir(dirString)* function.

Each packet of coordinates is stored as tab separated values. The first 6 values represent respectively the X,Y,Z,A,B,C coordinates of the robot tool central point (TCP) or the axial coordinates (A1,A2,A3,A4,A5,A6), depending from the value of "val" being equal to 1 or 2. The 7th and 8th values are the coordinates of up to external axes (if available). The 9th value is the ITRA timestamp (in microseconds).

If *val* is equal to 3, each line of the resulting text file will contain 15 values. The first 6 values represent respectively the X,Y,Z,A,B,C coordinates of the robot tool central point (TCP), the following 6 values are the axial coordinates (A1,A2,A3,A4,A5,A6). The 13th and 14th values are the coordinates of up to external axes (if available). The 15th value is the timestamp (in microseconds).

```
void getCurrPos(int robIndex, double *feedbackPos);
```

The function returns to the referenced array *feedbackPos* the most recent positional feedback received from the robot identified by index equal to "robIndex". *FeedbackPos* is an array of 15 double precision values. The first 6 values represent respectively the X,Y,Z,A,B,C coordinates of the robot tool central point (TCP), the following 6 values are the axial coordinates (A1,A2,A3,A4,A5,A6). The 13th and 14th values are the coordinates of up to two external axes (if available). The 15th value is the ITRA timestamp (in microseconds).

To call the function from MATLAB, *libpointer* has to be used:

```
feedbackPos = libpointer('doublePtr', zeros(15,1))       % initialize array of 10 elements
feedbackPos = calllib('robotComms','getCurrPos',0);      % call C function
```

You could also simply pass regular vectors and MATLAB takes care of marshalling:

```
feedbackPos = calllib('robotComms','getCurrPos',0,zeros(15,1));  % call C function
```

`double getTimestamp();`

The function returns a double precision number with the current ITRA timestamp. The returned number represents microseconds. The time is taken from the same clock used to timestamp the robot packets.

`bool isRSIRunning(int robIndex);`

The function checks if the latest packet is less than 12ms late. In this case the RSI is deemed as running and the function returns "true", otherwise it returns "false". The input (robIndex) is the robot index: 0 for Robot #1, 1 for Robot #2, and so on.

`bool isRobotTaskActive(int robIndex);`

The function checks if the latest packet is less than 12ms late. If it is the case, it checks if the latest packets contains $OUT[12]=="true". This causes the function to return "true". Otherwise the function returns "false". It has been decided that $OUT[12] will be used to mark the end of the RSI initialization from the robot program. The argument is the robot index: 0 for Robot #1, 1 for Robot #2 and so on.

`bool isDataAcquRequired(int robIndex);`

The function checks if the latest packets received from the Robot identified by *robIndex* contains $OUT[13]=="true". This causes the function to return "true". Otherwise the function returns "false". It has been decided that $OUT[13] will be used to mark the reach of a static robot pose where data acquisition is required. The input is the robot index: 0 for Robot #1, 1 for Robot #2 and so on.

`bool isRobStill(int robIndex);`

The function checks if the latest packet is less than 12ms late. If it is the case, it checks if the latest packets contains $OUT[13]=="true". This causes the function to return "true". Otherwise the function returns "false". It has been decided that $OUT[13] will be used to mark the reach of a static robot pose. The argument is the robot index: 0 for Robot #1, 1 for Robot #2 and so on.

`bool isRobMoveRequired(int robIndex);`

The function checks if the latest packet is less than 12ms late. If it is the case, it checks if the latest packets contains $OUT[14]=="true". This causes the function to return "true". Otherwise the function returns "false". It has been decided that $OUT[14] will be used by a robot to indicate allowance to move to the next pose is required. The argument is the robot index: 0 for Robot #1, 1 for Robot #2 and so on.

```
void setCartPos(int robIndex, double x, double y, double z, double a, double b, double c, double speed, double acc, int mode);
```

The function is designed to enable the external control of the robotic arm relative to "robIndex" through setting a target position (in the Cartesian reference system). The behaviour of the robot depends on the RSI configuration (see application examples below). "speed" and "acc" must be defined respectively in [mm/s] and [mm/s^2] if the function is used in conjunction with the RSI-Based External Control Approach or in percentage of the maximum joint speed and acceleration when the function is used in conjunction with the KRL-Based External Control Approach. The "mode" variable controls the way the target coordinates are transferred to the robot. With mode==1, the command packets are directly sent to the robot controller, otherwise (mode==0) it is appended to the list of previously generated command packets.

The function can also be used to support the KRL-based external control approach (see application examples below).

```
void setAxialPos(int robIndex, double a1, double a2, double a3, double a4, double a5, double a6, double speed, double acc, int mode);
```

The function is designed to enable the external control of the robotic arm relative to "robIndex" through setting a target position (in the Axial/Joint reference system). The behaviour of the robot depends on the RSI configuration. "speed" and "acc" must be defined as percentage of the maximum speed and acceleration of the robot, both in the RSI-Based External Control Approach and in the KRL-Based External Control Approach. The "mode" variable controls the way the target coordinates are transferred to the robot. With mode==1, the command packets is directly sent to the robot controller, otherwise (mode==0) it is appended to the list of previously generated command packets.

```
void setToolPathFromFile(int robIndex, string strPath, int mode);
```

The function is designed to enable the external control of the robotic arm relative to "robIndex" through sending a pre-generated command tool-path to the robot (in the Cartesian reference system). This function can be used only in conjunction with the Computer-Based external control apporach. "strPath" is the path of the ".txt" file containing the path command packets. The "mode" variable defined for *setAxialPos* and *setCartPos* is not available for this function. The function always appended the command packets contained in the text file to the queue of command packets the RSI Manager sends to the RSI interface.

```
void allowRobotStart(int robIndex);
```

The function is developed to perform the following consecutive actions:

- send a flag to the robot to set $SEN_PINT[11] to "true"
- wait for $OUT[11] to become "false"
- send a flag to the robot to set $SEN_PINT[11] to "false"
- wait for $OUT[11] to become "true"

It has been decided that the actions above will match corresponding actions on the robot program to allow the robot to start its pre-programmed task. The argument of the function is the robot index: 0 for Robot #1, 1 for Robot #2 and 2 for Robot #3.

```
void allowRobMove(int robIndex);
```

The function is developed to perform the following consecutive actions:

- send a flag to the robot to set $SEN_PINT[12] to “true”
- wait for $OUT[13] to become “false”
- send a flag to the robot to set $SEN_PINT[12] to “false”

It has been decided that the actions above will match corresponding actions on the robot program to allow the robot to move to the next pose of its pre-programmed task. The argument of the function is the robot index: 0 for Robot #1, 1 for Robot #2 and so on.

```
void requRobTaskEnd(int robIndex);
```

The function is developed to perform the following consecutive actions:

- send a flag to the robot to set $SEN_PINT[13] to “true”
- wait for $OUT[14] to become “true”

It has been decided that the actions above will match corresponding actions on the robot program to allow the robot to approach the end of the pre-programmed task. The argument of the function is the robot index: 0 for Robot #1, 1 for Robot #2 and so on.

```
void requRealTimeEnd(int robIndex);
```

The function is developed to exit the KRL RSI_MOVECORR() line in the RSI-based and in the Computer-based external control. The function performs the following consecutive actions:

- send a flag to trigger the STOP object in the RSI configuration and cause the termination of RSI_MOVECORR() line in the KRL programme;
- waits for $OUT[12] to become “false”

NOTE: the function never terminate if $OUT[12] is not set to “false” in KRL, after RSI_MOVECORR() terminates. The argument of the function is the robot index: 0 for Robot #1, 1 for Robot #2 and so on.

```
void allowRobotFinish(int robIndex);
```

The function is developed to perform the following consecutive actions:

- send a flag to the robot to set $SEN_PINT[11] to “true”
- wait for $OUT[11] to become “false”
- send a flag to the robot to set $SEN_PINT[11] to “false”
- wait for $OUT[11] to become “true”

It has been decided that the actions above will match corresponding actions on the robot program to allow the robot to return to the home position and close the RSI context. The argument of the function is the robot index: 0 for Robot #1, 1 for Robot #2 and so on.

```
void terminateRSIManager();
```

The function terminates the RSIManager thread and waits that the queue of packets waiting to be saves is emptied. Therefore, also the Saving thread is terminated.

External control application examples

Introduction

Robots have been quite successful in accomplishing tasks in well-known environments like a work cell within a factory. The much harder problem of a robot acting in unstructured and dynamic environments, like those humans normally act and live in, is still an open research area [6]. In such situations, the robots need to adapt their tasks. Real-time robot motion control can be divided into two subproblems: (i) the specification of the control points of the geometric path (path planning), and (ii) the specification of the time evolution along this geometric path (trajectory planning). This section presents the application of ITRA to achieve external control of robotic arms. Three different approaches are presented.

Whereas the path-planning subproblem is always dealt with by the computer hosting ITRA, where processing of machine vision data and/or other sensor data can take place to compute the robot target position, the trajectory planning subproblem can be managed by different actors of the system. In the first approach (hereafter referred as KRL-based approach), the trajectory planning takes place at level of the KRL module running within the robot controller. The second approach has trajectory planning performed within the external computer, soon after path-planning, and is referred as Computer-based approach. The third approach relies on a real-time trajectory planning algorithm implemented into the RSI configuration. Therefore, trajectory planning is managed by the RSI context and the approach is named as RSI-based approach. This section contains the instructions to setup the three external control approaches and test them through interactive Matlab-based graphical user interfaces (Fig. 3), through which the user can jog the robot moving the mouse hover interactive 2D plots. The Matlab interface is just to generate command target positions to send to the robot. Using such examples, the user will understand how to use basic ITRA functionalities and its external control functionalities. The trained user will be able to replace the example Matlab interface with the required algorithms to generate the control points.

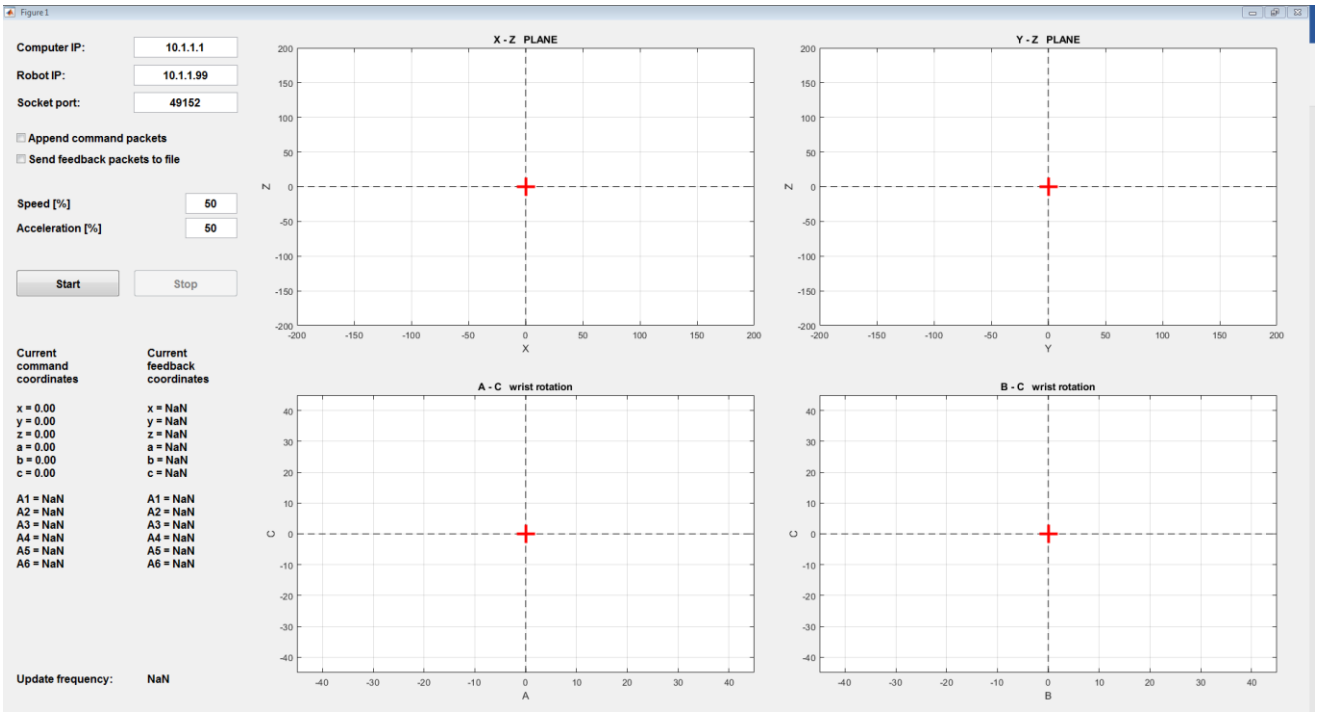

Fig. 3. Matlab-based graphical user interfaces for external control example applications.

Preliminary installation steps

To correctly run the application examples, the following steps must be followed:

1. Download the example files coming with the present reference manual.
2. Copy the ITRA library files (*robotComms.dll/robotComms.so* and *robotComms.h*) to the same folder, within the PC, hosting the Matlab example script.
3. Copy the XML schema (ITRA.xml)(Fig. 2) into the robot controller, in the following directory: C:\KRC\Roboter\Config\User\Common\SensorInterface
4. Copy the required RSI-Visual configuration (depending on the example to run) into the robot controller, in the following directory:
   C:\KRC\Roboter\Config\User\Common\SensorInterface
   a. Note, a valid RSI-Visual configuration consists of the RSI, DIAGRAM and XML files. They form a unit and must be transferred to the robot controller together.
   b. <File name>.rsi: signal flow configuration from RSI Visual
   c. <File name>.rsi.diagram: signal flow layout from RSI Visual according to XML schema
   d. <File name>.rsi.xml: XML file for signal processing on the robot controller
5. Copy the file provided KRL module <filename>.src (depending on the example to run) inside the specific program folder of the KUKA System Software (KSS) in the KRC (e.g. C:/KRC/ROBOTER/.../R1/Program/).

KRL-based approach

This approach is based on the use of ITRA in conjunction with the RSI configuration shown in Fig. 4a. This configuration maps the eight double precision values available at the outputs of the ETHERNET object to the first eight elements of an array of real numbers ($SEN_PREA), which is accessible from the KRL module. The ITRA setCartPos function can be used to send the target Cartesian space coordinates (X, Y, Z, A, B, C) and the desired speed and acceleration with which the target must be reached. The coordinates are mapped to $SEN_PREA[1-6], the speed gets mapped to $SEN_PREA[7] and the acceleration to $SEN_PREA[8]. Therefore, the array element values can be assigned to local variables in the KRL module and the target position can be reached through a point-to-point (PTP) movement within a loop structure. The KRL code responsible for extracting the values stored in $SEN_PREA and moving the robot is the following:

```
» LOOP
»       target_pos.x = $SEN_PREA[1]
»       target_pos.y = $SEN_PREA[2]
»       target_pos.z = $SEN_PREA[3]
»       target_pos.a = $SEN_PREA[4]
»       target_pos.b = $SEN_PREA[5]
»       target_pos.c = $SEN_PREA[6]
»       $VEL.CP = $SEN_PREA[7]/1000
»       $ACC.CP = $SEN_PREA[8]/1000
»       PTP target_pos
»       IF ($SEN_PINT[13]==1) THEN
»             $OUT[14]=TRUE
»             EXIT
»       ENDIF
» ENDLOOP
```

In this example the coordinates are given in millimeters, whereas desired speed and acceleration are respectively given in mm/s and mm/s^2. Modifying the KRL code, it is possible to use linear (LIN) movements, rather than PTP movements.

To test this approach, the user has to:

1. Open the "KRL Approach" folder contained in the "Examples" folder;
2. Copy the files into *"..\KRL Approach\RSI Configuration"* into the robot controller, in the directory *"C:\KRC\Roboter\Config\User\Common\SensorInterface"*
3. Copy the files into *"..\KRL Approach\KRL Module"* inside the user specific program folder of the KUKA System Software (KSS) in the KRC (e.g. *C:/KRC/ROBOTER/.../R1/Program/*).
4. In T1 mode, make sure the home position defined in the KRL module is suitable for the robot in use.
5. Connect robot controller to external computer making sure you can ping the robot from the computer and the computer from the robot controller;

6. Run the KRL module in T1 mode;
7. Run the Matlab example in *..\KRL Approach\Matlab test programme* and test external control.

It is also easy to customize the KRL code to control the robot through joint space coordinates (A1-A6) rather than Cartesian coordinates. In this case, the setAxialPos function should be used on the computer side. The external control can be terminated through the requRobTaskEnd function that sends a Boolean flag to set $SEN_PINT[13] true and waits for $OUT[14] to became true too. Since the robot controller interprets the KRL module line by line, one limitation of this approach is that the robot must decelerate and stop at the target position. This is to allow the KRL interpreter to return to the beginning of the loop and extract the new target coordinates, and the required speed and acceleration from the $SEN_PREA array. This approach is unable to provide true real-time control of the robot, since a previously commanded target must be reached, before a new target position can be assigned. Moreover, only PTP and LIN interpolations are available.

Computer-based approach

This second approach is based on the use of ITRA in conjunction with the RSI configuration shown in Fig. 4b. Here the target coordinates available at the output of the ETHERNET object are given to the inputs of the POSCORR object, which allows Cartesian correction of the robot position within a range (limits specified in the object parameters). POSCORRMON is the object that limits the maximum overall Cartesian correction. The KRL code responsible for activating the motion guided by the external computer is reduced to the following line:

```
» RSI_MOVECORR()
```

Once the KRL interpreter reaches this line, the robot drives start actuating the target coordinates received by the RSI context at every cycle; the KRC is made no longer responsible for planning the kinematics and dynamics of the trajectory used to reach the target. Therefore, it is crucial the communication between the computer and the RSI context is stable and no command packets are lost. Moreover, it is important the commanded trajectory is smooth and the associated velocity and acceleration patterns are continuous. Methods to compute control points for smooth trajectories have been presented in [7]. The setToolPathFromFile ITRA function supports this external control approach, enabling the possibility to send all trajectory control points with no delays. The function is called giving, as inputs, the index of the robot to control and the name of a text file, where all target positional packets are stored in advance. The function accesses the text file and loads all command packets into the ITRA command queue relative to the robot to control. Each packet is promptly dequeued and sent to the RSI context by the ITRA RSI-Manager Thread, which guarantees all packets are sent sequentially and each packet is sent within the RSI cycle duration. This external control approach allows executing a trajectory, sending a control point per each interpolation cycle of the robot controller. Therefore, this approach is ideal to follow complex trajectories accurately. In the Computer-approach, the external control can be terminated through the requRealTimeEnd function; it transmits a Boolean flag that triggers the STOP object in the RSI configuration. This causes the KRL interpreter to terminate the RSI_MOVECORR() line.

In a similar way to the KRL-based approach, the limitation of the Computer-based approach is that it is necessary to wait all trajectory points are sent, before a new set of points can be streamed to the robot. The reason lies in the fact that a sudden interruption of the sequential transfer of control points would cause an immediate stop of the motion with a consequent peak/discontinuity in the velocity, acceleration and jerk.

To test this approach, the user has to:

1. Open the "Computer Approach" folder contained in the "Examples" folder;
2. Copy the files into *"..\Computer Approach\RSI Configuration"* into the robot controller, in the directory *"C:\KRC\Roboter\Config\User\Common\SensorInterface"*
3. Copy the files into *"..\Computer Approach\KRL Module"* inside the user specific program folder of the KUKA System Software (KSS) in the KRC (e.g. *C:/KRC/ROBOTER/.../R1/Program/*).
4. In T1 mode, make sure the home position defined in the KRL module is suitable for the robot in use.
5. Connect robot controller to external computer making sure you can ping the robot from the computer and the computer from the robot controller;
6. Run the KRL module in T1 mode;
7. Run the Matlab example in *"..\Computer Approach\Matlab test programme"* and test external control.

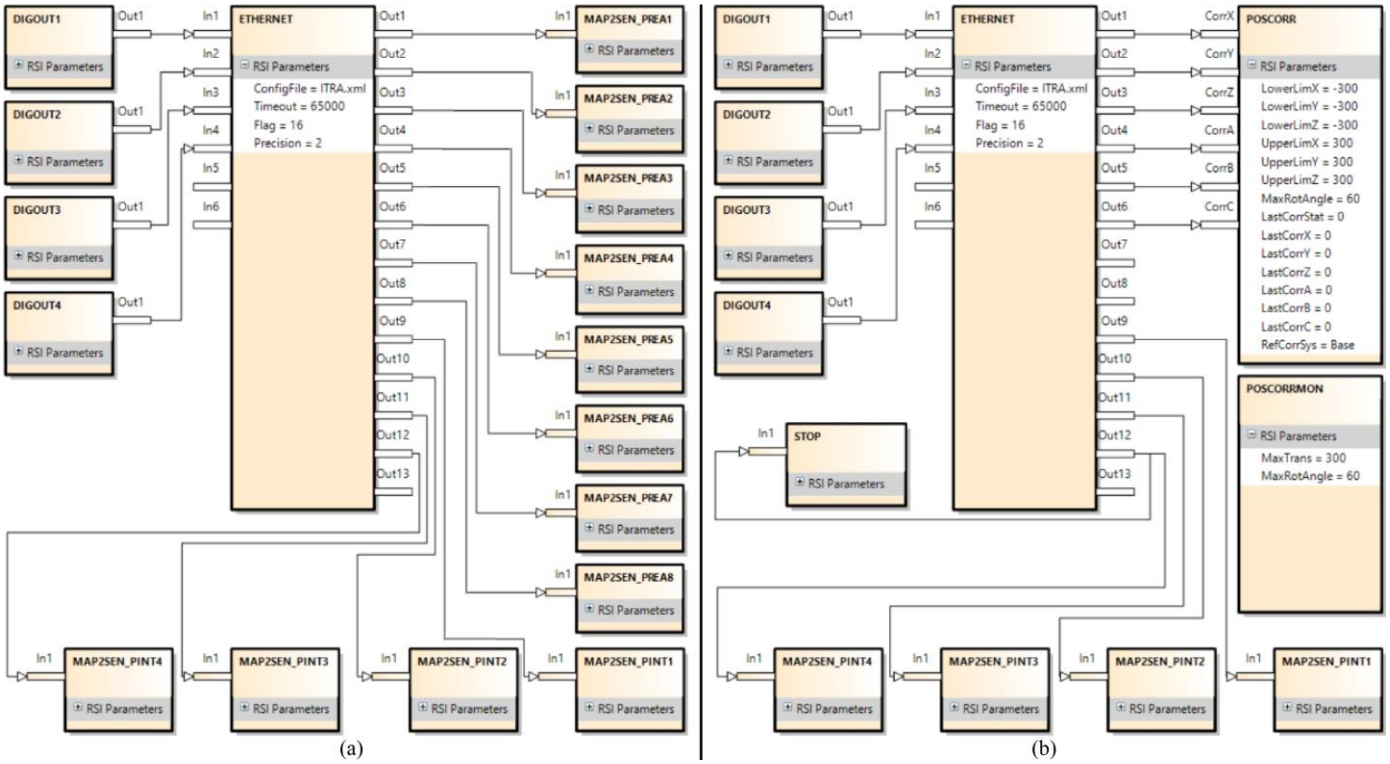

Fig. 4. RSI-Visual configurations for external path control: KRL-based (a) and Computer-based (b) approach.

RSI-based approach

Unlike the KRL-based and the Computer-based approaches, the RSI-based approach allows achieving true real-time path control of KUKA robots based on KRC4 controllers. This approach permits applying fast online modifications of planned trajectory, to adapt to changes in the dynamic environment and react to unforeseen obstacles. Whereas the path-planning takes place into the server computer, the trajectory planning has been implemented into a RSI configuration, employing the second-order trajectory generation algorithm presented in [8]. The approach can operate in Cartesian-space and in joint-space. Due to the complexity of the relative RSI-Visual configurations (containing over 500 objects), they are not shown herein but they are available in the ITRA software package containing the present manual. While the robot is static or travels to a given position, the computer can send a new target position (together with the maximum preferred speed and acceleration) though the *setCartPos* or the *setAxialPos* functions. Unlike the RSI configuration of the Computer-based approach, the target coordinates received into the RSI context are not passed to POSCORR object. Such target coordinates are instead used to compute the optimal coordinates of the set point to send to the object through a two-fold algorithm. On the one hand, the set point is generated to guarantee a smooth transition from the initial conditions (starting coordinates, velocity and acceleration) towards the final target position. On the other hand, the algorithm makes sure the evolution of the robot motion is constrained within the given maximum speed and acceleration.

To test this approach, the user has to:

1. Open the "RSI Approach" folder contained in the "Examples" folder;
2. Copy the files into "*..\RSI Approach\RSI Configuration/Axial"* and "*..\RSI Approach\RSI Configuration/Cartesian"* into the robot controller, in the directory "*C:\KRC\Roboter\Config\User\Common\SensorInterface"*
3. Copy the files into "*..\RSI Approach\KRL Module"* inside the user specific program folder of the KUKA System Software (KSS) in the KRC (e.g. *C:/KRC/ROBOTER/.../R1/Program/*).

4. In T1 mode, make sure the home position defined in the KRL module is suitable for the robot in use.
5. Connect robot controller to external computer making sure you can ping the robot from the computer and the computer from the robot controller;
6. Run the KRL module in T1 mode;
7. Run the Matlab example in *"..\RSI Approach\Matlab test programme"* and test external control.

Benchmarking

The run-time of all ITRA functions was investigated loading the DLL files into Matlab 2018a (64bit version), running within a computer with Intel i7-7700HQ CPU and 16GB of RAM. The computer, based on a Windows 10 64bit operating system, was linked to one KR6 R900 AGILUS robot running a KRL module that contained all required lines to enable the execution of the ITRA functions.

Function run-times

Each function was executed 100 times, to record the minimum, the maximum and mean value of its run time. Table II reports the resulting values in micro-seconds (µs). All functions were tested with the RSI context running at 4 ms cycle mode, with the exception of setToolPathFromFile, which is used for the Computer-based external control approach that is only supported by the 12 ms RSI cycle mode. The run-time of setToolPathFromFile depends on the number of command positions contained in the text file, which affects the time to load them into the DLL command queue. The function was tested with a file containing 250 positions.

TABLE II
RUN-TIME FOR ALL ITRA FUNCTIONS.

	Function names	Run time [µs] Min	Max	Mean
Initializers	*setNumRob*	9.48	587.48	24.58
	setRobIP	6.19	25.52	7.81
	setRobConnType	4.74	21.88	6.03
	setOutputDir	9.11	492.67	16.30
	setRobFeedbackOutput	4.01	26.98	5.74
Networking	*openConn*	68.92	175.04	91.03
	isDataAvailable	6.92	28.07	9.17
	startRSIManager	938.66	2607.03	1185.81
	terminateRSIManager	7.29	27.71	8.75
	closeConn	29.17	71.84	49.48
Getters	*isRSIRunning*	13.12	59.07	26.55
	isRobotTaskActive	5.10	28.80	11.99
	isRobStill	5.12	28.75	12.04
	isRobMoveRequired	30.99	100.64	55.85
	isDataAcquRequired	5.10	28.80	9.02
	getCurrPos	33.18	90.07	45.79
	getTimestamp	6.19	25.16	8.25
Setters	*allowRobotStart*	25279.76	27070.66	26722.90
	allowRobMove	8107.01	11959.04	10333.85
	allowRobotFinish	23190.56	24114.27	23976.75
	requRealTimeEnd	92377.52	104013.83	98249.93
	requRobTaskEnd	11819.00	12772.98	11981.29
	setCartPos	10.94	65.27	24.80
	setAxialPos	10.95	65.26	24.81
	setToolPathFromFile	5516.38	11290.96	7414.96

External control reaction times

The performance of the three external control approaches was also tested. Reaction time is the most important parameter in real-time control, since it measures the promptness of the system. Reaction time in humans is a measure of the quickness the organism responds to some sort of stimulus. The reaction time is defined as the latency between the stimulus and the very start of the reaction. The average reaction time for humans is 250ms to a visual stimulus, 170ms for an audio stimulus, and 150ms for a touch stimulus [9]. The reaction speed plays a large part in everyone's everyday life. Fast reactions can produce big rewards, for example like saving a blistering soccer ball from entering the goal. Slow reaction times may come with consequences. Similarly,

achieving small reaction time is crucial for robots that need to have real-time adaptive behaviors to respond to dynamic changes and/or to interactions with humans.

The external control latency (or reaction time) is defined herein as the time interval between the instant a new target position becomes available on the external computer and is sent to the robot via setToolPathFromFile, setCartPos or setAxialPos and the instant the robot starts reacting to reach such commanded target. With ITRA running within Matlab and saving robot feedback positions through the saving thread, the reaction time of each external control approach was measured 100 times through commanding the robot to move to a target from a static position. The timestamp of the first robot feedback positional packet, reporting a deviation greater or equal to 0.01mm from the original home position, was compared with the timestamp taken by getTimestamp just before sending the target position to the robot. The resulting reaction times are given in Table III.

TABLE III
PERFORMANCE OF EXTERNAL CONTROL APPROACHES

External control approach	RSI cycle	Update rate	Reaction time [ms]		
			Min	Max	Mean
KRL-based	4 ms	Variable	48.61	175.73	113.44
Computer-based	12 ms	Variable	54.86	81.55	64.84
RSI-based	4 ms	250 Hz	27.18	31.99	30.03

The average robot reaction time given by the three approaches is always better than the human reaction time, when responding to touch stimulus. The first approach is 23% better than the human reaction. The second and the third approach are respectively 57% and 80% better. The update rate of the first and second approach is variable, since a new target position can be commanded only after the previous target is reached. The update rate of the RSI-based approach is equal to the running frequency of the RSI context, so a new target position can be set every 4 ms with the robot expected to react within 30 ms (±3 ms).

REFERENCES

[1] KUKA, *KUKA.RobotSensorInterface 3.2 Documentation - Version: KST RSI 3.2 V1*. 2013.
[2] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of VxWorks, Linux, RTAI and Xenomai in a hard real-time application," in *Real-Time Conference, 2007 15th IEEE-NPSS*, 2007, pp. 1-5: IEEE.
[3] B. Stroustrup, *The C++ programming language*. Pearson Education, 2013.
[4] K. D. Nilsen, "Reliable real-time garbage collection of C++," *Computing Systems,* vol. 7, no. 4, pp. 467-504, 1994.
[5] K. Houstoun and E. Briggs, "Rapid addition leverages Microsoft .NET 3.5 Framework™ to build ultra-low latency FIX and FAST processing," ed, 2014.
[6] T. Kunz, "Real-Time Motion Planning for a Robot Arm in Dynamic Environments," Verlag nicht ermittelbar, 2009.
[7] C. Mineo, S. G. Pierce, P. I. Nicholson, and I. Cooper, "Robotic path planning for non-destructive testing–A custom MATLAB toolbox approach," *Robotics and Computer-Integrated Manufacturing,* vol. 37, pp. 1-12, 2016.
[8] R. Haschke, E. Weitnauer, and H. Ritter, "On-line planning of time-optimal, jerk-limited trajectories," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, 2008, pp. 3248-3253: IEEE.
[9] G. R. Grice, R. Nullmeyer, and V. A. Spiker, "Human reaction time: toward a general theory," *Journal of Experimental Psychology: General,* vol. 111, no. 1, p. 135, 1982.