WAM^{TM} Arm

Customer Training Documentation

Barrett Technology™ Inc.

Contents:

Barrett Technology Support:

WAM Maintenance:

Libbarrett:

Barrett Technology Support

Description:
This section describes resources for future support working Barrett Technology's products the WAM and BarrettHand 280.

Accessing Barrett Technology Support Download Area

The support download area is included with your customer support subscription. This download area contains the latest resources including documentation, video manuals, demonstrations, software, firmware and more. These resources are constantly updated by the manufacturing and engineering team at Barrett Technology. Upon expiration of your support subscription, a snapshot of the support site will be taken. From that point on, the resources will be available forever, but they will not be updated with the latest releases.

The support download area is available any time at the address:
http://web.barrett.com/support

You may log in with the credentials unique to each customer.

Support subscription code:

CODE: <<*code*>>

Accessing Barrett Technology Support Wiki

The support wiki is another great resource for your usage and development with the WAM and BarrettHand. This area is readily accessible to the public, requiring no credentials to access and contains standard product support information. These resources are constantly updated by the manufacturing and engineering team at Barrett Technology.

The Barrett Technology support wiki is accessible at:
http://support.barrett.com

Contacting Barrett Support by Email

Barrett Technology takes pride in our customer support, particularly through our communications via email with customer support representatives. Responses and solutions are expected in one to two business days. If you find that you cannot resolve an issue using the support wiki or download area, please contact us at any time with an email to support@barrett.com.

WAM Maintenance

Description:
This section describes maintenance routines for the WAM and BarrettHand 280. These will all be achieved through existing software solutions.

WAM Calibration

Calibration of the WAM is necessary for position and gravity compensation accuracy using Libbarrett.

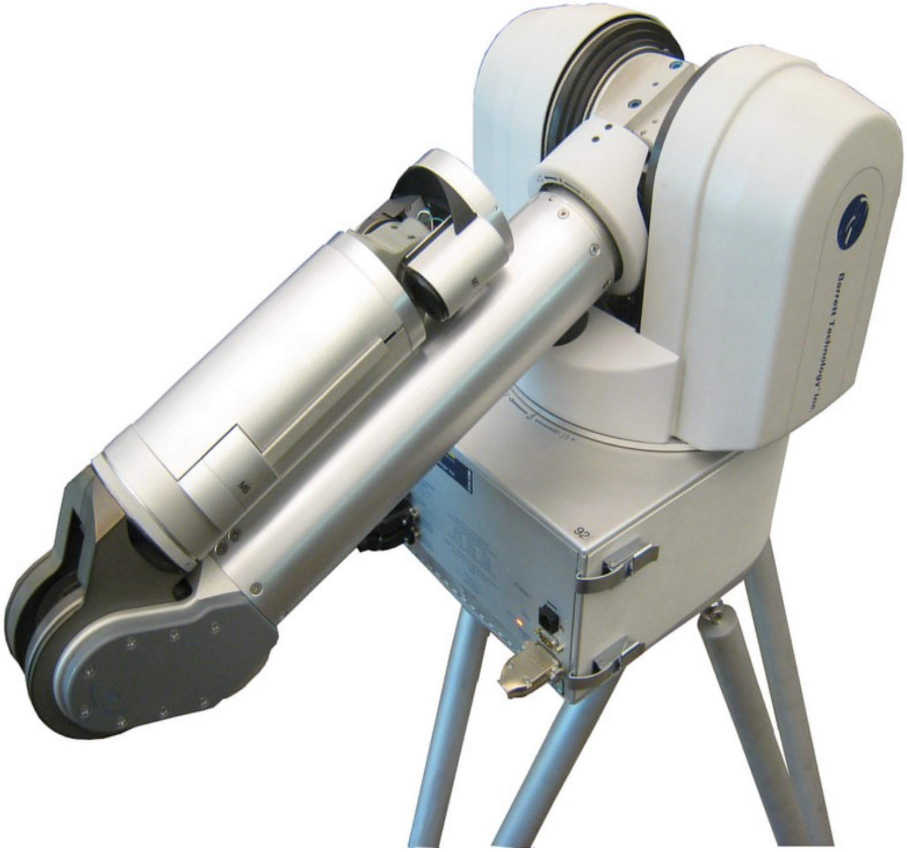
Zero Calibration

The WAM uses absolute encoders at the each motor for control of the arm. Because of the transmission ratios seen below, the robot must be placed in a known configuration allowing us to know the exact position of the WAM arm (typically within 5 to 10 degree, or one motor revolutions. This position is known as the WAM's home position (standard home position seen below). Variance is introduced by different 'home' positions. Zero calibration is a routine to find the offsets that describe the exact zero radian position for each joint from the home position.

Arm Transmission Ratios

Parameter	Value
N_1	42.0
N_2	28.25
N_3	28.25
n_3	1.68
N_4	18.0
N_5	9.48
N_6	9.48
N_7	14.93

Standard Home Position:



Running Zero Calibration (bt-wam-zerocal)

The bt-wam-zerocal routine should be run only upon one of the following:

- New Installation
- Moving the WAM to a new location
- After replacing a cable
- After tensioning a cable
- After replacing a puck
- After a purge of the previous calibration data (/etc/barrett/) for any reason

bt-wam-zerocal can be run with the following command:

```
$ bt-wam-zerocal
```

Please follow the on-screen instructions to zero each joint.

A visual description of the correct joint positions can be found:
http://web.barrett.com/support/WAM_Documentation/zero_calibration_procedure.pdf

Gravity Calibration

Standard Libbarrett software presents an easy interface for gravity compensation of the WAM and attached end-effector. Gravity compensation requires the running of a gravity calibration routine to determine the forces necessary to compensate the weight of each link.

The bt-wam-gravitycal routine should be run only upon one of the following:

- A new device is attached to the end-effector
- Any changes to the mounted hardware
- Any of the previous cases described requiring bt-wam-zerocal

bt-wam-gravitycal

bt-wam-gravitycal can be run with the following command:

```
$ bt-wam-gravitycal
```

Please follow the on-screen instructions to run the calibration routine. The calibration will go through 5 poses for the 4-DOF WAM configuration or 9 poses for the 7-DOF WAM configuration to calculate the optimal torques necessary for gravity compensation.

WAM Cable Tensioning

Keeping tension in the WAM cables in necessary to achieve control accuracy, and also to elongate the expected lifetime of the cables. Barrett Technology suggests tensioning the WAM cables after 80 hours of typical WAM usage.

Autotensioning with Libbarrett

A new autotensioning routine (bt-wam-autotension) has been released using Libbarrett. This software will pull the slack introduced into the system from extended usage.

As of the Libbarrett-1.2.0 release, this is an installed routine, similar to bt-wam-zerocal and bt-wam-gravitycal.

bt-wam-autotension can be run with the following command:

```
$ bt-wam-autotension
```

Arguments can also be passed to this program to tension specific joints (ex. 1,3, and 6)

```
$ bt-wam-autotension 1 3 6
```

WAM/BarretHand Puck Firmware

Puck firmware updates are periodically released. It is advised that customers are always running the latest available puck firmware.

Manual Firmware Update with Btclient

Move to the btutil directory

```
$ cd ~/btclient/src/btutil
```

Press the control pendant E-STOP button.

Release the control pendant E-STOP button.

Press <SHIFT+IDLE> on the control pendant.

```
$ ./btutil -d <id> -f puck2.tek.rXXX
```

Wait for the download to complete

Set the default parameters for the new firmware (described on page 12)

Automatic Firmware Update with Shell Script

A simple bash script has been created to automate the motor puck firmware update process. This script is updated with each firmware release.

Download the script

```
$ wget http://web.barrett.com/support/WAM_Installer/update_wam_firmware.sh
```

Run the script

```
$ bash update_wam_firmware.sh
```

Follow the on-screen instructions to ensure a successful firmware update.

After a successful firmware update, we suggest finding new motor offsets (described on page 13).

Automatic Set WAM Default Puck Properties with Shell Script

A simple bash script has been created to automate the setting of default properties for WAM motor pucks. It is necessary to run this script after manual firmware updates. The automated firmware update script includes the setting of default properties.

Download the script:
$ wget http://web.barrett.com/support/WAM_Installer/set_wam_default_properties

Set WAM Puck Properties in C++ with Libbarrett

Libbarrett also makes it possible to set puck properties. Lists of puck properties and their default values can be found on the support download site. Puck properties for the current motor firmware (r200) can be found:
http://web.barrett.com/support/Puck_Documentation/PuckProperties-r200.pdf

Example program – Setting puck properties in Libbarrett.

```
int main(int argc, char** argv) {
  ProductManager pm; // Instantiate our product manager
  for(size_t i=11;i<15;i++){ // Step over each hand puck (11-14)
    Puck* puck = pm.getPuck(i);
    printf(“Puck %d TSTOP Property currently set to: %d/n”, i,
           puck->getProperty(Puck::TSTOP);
    printf(“Setting TSTOP to 100”);
    puck->setProperty(Puck::TSTOP, 100);
    printf(“Puck %d TSTOP Property now set to: %d/n”, i,
           puck->getProperty(Puck::TSTOP);
  }
}
```

Find Motor Puck Offsets

After replacing a puck, it is necessary to find two offsets (IOFST and MOFST) using this procedure. IOFST is a "current offset", the current sensor value when zero current is commanded. MOFST is a "motor offset", the offset between the encoder's zero value and the start of an electrical cycle in the motor.

Do this for each WAM joint.

Move the joint to the center of its range.

Enter the btutil directory.

```
$ cd ~/btclient/src/btutil
```

Press the control pendant E-STOP button.

Release the control pendant E-STOP button.

Press <SHIFT+IDLE> on the control pendant.

```
$ ./btutil -f
```

Select motor to calibrate.

Wait 10 seconds. Do not touch the robot.

After the calibration has completed, press the control pendant E-STOP, release the control pendant E-STOP and press <SHIFT+IDLE>. This will load the newly calibrated values.

Backup CF Card

Instructions for backing up your current CF card by making an exact image for the disk.

These instructions require a CF to USB adapter.

Install fsarchiver

```
$ sudo apt-get install fsarchiver
```

Insert CF card and adapter into your PC. Close all pop-up windows showing the contents of the CF card.

Download the automated backup script.

```
$ wget http://web.barrett.com/support/WAM_Installer/backup_cf.sh
```

Run the script

```
$ bash backup_cf.sh
```

Follow the on-screen instructions to complete the backup process. An image of your CF card will now be found in the current directory.

Create New CF Card

If a previous CF card is damaged or a new OS release is made available by Barrett, you may want to create a new CF card. This automated CF card creation script will be updated with any such new OS releases.

These instructions require a CF to USB adapter.

Download fsarchiver

```
$ sudo apt-get install fsarchiver
```

Download the automated CF card installation script

```
$ wget http://web.barrett.com/support/WAM_Installer/create_cf.sh
$ bash create_cf.sh
```

Follow the on-screen instructions to create the Ubuntu Xenomai 12.04 CF Card.

With the power off, insert the new CF image into the WAM's PC/104. Boot the WAM with the new OS.

You should now be able to SSH into the WAMs PC/104. Note: You may need to remove your ~/.ssh/known_hosts file as the ssh_key will be different.

On the 12.04 CF Image home directory is a file - install_software.sh. This file will install the latest libbarrett and btclient software.

```
$ bash install_software.sh
```

Libbarrett

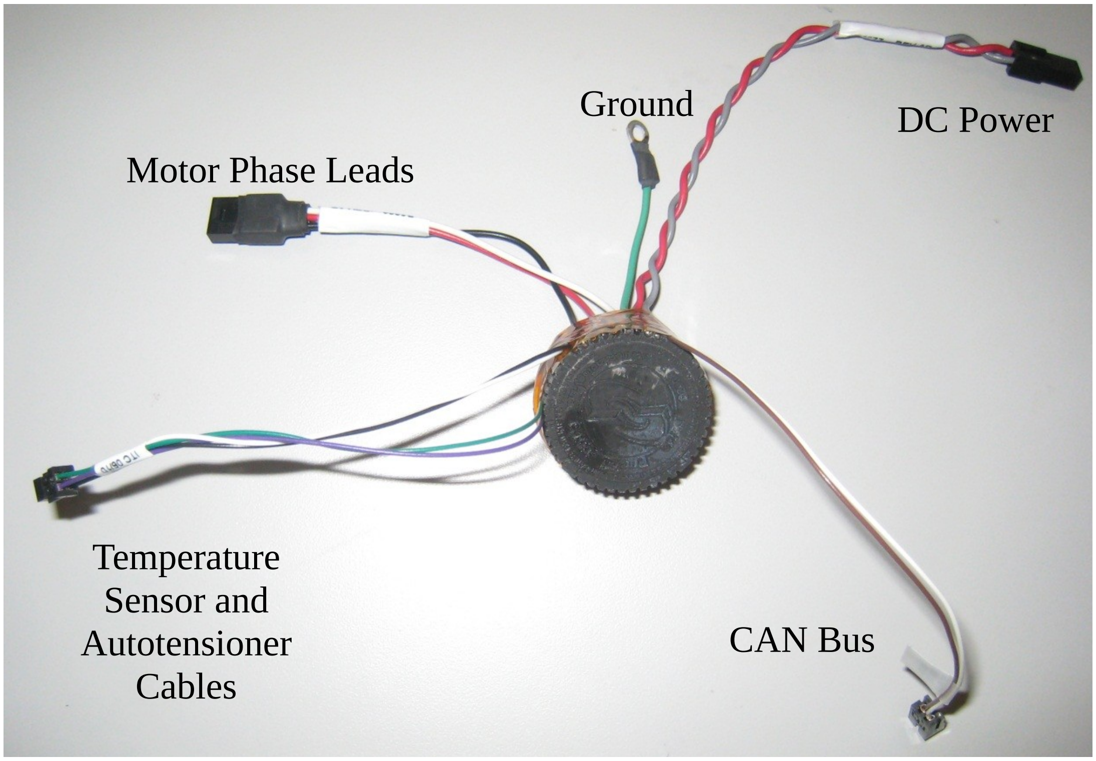
Description:
This section describes Libbarrett, Barrett Technology's real-time robot control library written in C++.

Libbarrett Introduction

The WAM arm is made of high precision machined aircraft-grade aluminum. It is a 'cable-driven' arm. This unique actuation technique eliminates the need for conventional gears, resulting in a highly back-drivable system which is easily manipulated. The WAM is a torque controlled robot. Because of the direct-drive capabilities, the WAM allows for extremely graceful motions and robust control of contact forces, independent of mechanical force or torque sensors. The WAM has a modular design, allowing for four or seven degree-of-freedom configurations with or without the use of the BarrettHand 280. A self contained PC and safety system make the WAM arm a small package ideal for mobile manipulation and transportability.

The WAM's motors are driven by Pucks, Barrett Technology's patented motor controllers. Pucks are exceptionally small and are placed throughout the WAM Arm in-line with the motors, rather than stored in a motor controller box outside of the robot. Pucks communicate with the control PC through the CAN Bus (Controller Area Network), with all communications monitored by the Safety Board. The Safety Board is a large PCB in the base of the WAM that monitors the velocity of the arm, the magnitude of the torque commands being sent to the motors, the communication rate between the control PC and the Pucks, and the status of the buttons on the Pendants. The CAN bus is a two-wire differential serial bus that provides digital communication at 1 Mbps with high noise immunity. The protocol used is proprietary for the WAM (not CANopen). For more information on CAN, visit http://en.wikipedia.org/wiki/CAN_bus. Pucks are the lowest level of control that the programmer needs to understand. The motor pucks are capable of receiving motor torque commands. A real-time program is expected to command these torques at a specified control rate. This control loop may run at any rate up to 1 kHz, but the default rate is 500 Hz. The real-time control program must send a torque command at this frequency, translating it into the proprietary CAN signal and address it to the appropriate Pucks. The image below shows a Puck and its leads.

A Motor Puck



Libbarrett is an open-source (GPL license), real-time controls library written in C++ for control of Barrett Technology's products including the WAM, the Barrett Force/Torque Sensor and BH8-280 BarrettHand. Libbarrett provides a convenient API for control of these products from the highest to lowest levels. The library runs on Linux operating systems with Xenomai 2.6.1 using Xenomai's RT-Socket-CAN Driver. The Library has been tested using Ubuntu 9.10, 10.04, and 12.04. With the exception of a few minor maintenance tasks (shown above), Libbarrett replaced the older btclient controls library, which is no longer under active development.

A few of Libbarrett's most important features for running a WAM include:

- Real-time joint torque control
- Real-time sensor-less force/torque control (Cartesian torque control / Haptics)
- Real-time joint position/velocity control
- Real-time Cartesian position/velocity control
- Real-time tool orientation position/velocity control
- Real-time pose (position & orientation) position/velocity control
- Real-time data logging capabilities

- Access to the Jacobian matrix
- User-calibrated gravity compensation
- Simple interface for point-to-point moves and spline-following in both joint and Cartesian space
- Teach-and-play
- Forward kinematics calculations
- Online inverse kinematics (based on Jacobian-transpose methods)
- Inverse dynamics calculations

Using Xenomai, Libbarrett uses real-time code in order to execute commands properly. Real-time computing guarantees that operations will occur at a given frequency, allowing for precisely timed code without interference from standard operating system processes which delay the clock and skewing time readings. Each Libbarrett control cycle consists of asking for the pucks' motor positions, the pucks sending their positions, Libbarrett's real-time computations necessary to send new motor torques, and the sending of the new motor torques to the pucks. It is extremely important that software written using Libbarrett does not get in the way of this timing, or else the robot will fault with the reading 'heartbeat fault' due to a lag in the communications and a control cycle being missed. Precautions and monitoring of the system log (/var/log/syslog) for monitoring your system's mean, max, etc. cycle times, are useful for ensuring that your real-time code is meeting the timing requirements of your specified control rate.

Libbarrett control takes place in 'system' based components. This is akin to MathWorks Simulink software's systems, Boost Systems, or Qt's Signals and Slots. It can be thought of as independent processes with specified input and output data types. More information can be found on page 20.

Introduction to Barrett Technology's Real-Time Operating System

Libbarrett relies on Xenomai for its guaranteed execution of real-time code. Barrett Technology compiles vanilla kernels, with Xenomai patches, and the necessary Ubuntu patches resulting in our real-time capable kernels. Our latest release is Ubuntu 12.04 with Xenomai 2.6.1. Compiled into the kernel are the necessary drivers (RT-Socket-CAN) for the PEAK CAN cards in usage on Barrett's products.

Xenomai (http://www.xenomai.org) provides an API for real-time development. Using the capabilities of hardware sharing exposed by the Adeos project and its patches (http://home.gna.org/adeos/), Xenomai provides hard real-time support to user-space applications.

For further information on Xenomai, the white paper is available at:
http://www.xenomai.org/documentation/branches/v2.3.x/pdf/xenomai.pdf

For further information on Adeos, the white paper is available at:
http://opersys.com/ftp/pub/Adeos/adeos.pdf

Overview of Libbarrett Real-Time Systems

Libbarrett uses 'system' based components for control of the WAM and BarrettHand 280. The library comes with many built-in real-time systems for control of the WAM and BarrettHand, and allows users to create their own real-time systems.

Upon instantiation of the WAM object, a templated class based on the DOF of the WAM, all of the systems and other objects necessary for real-time control of the WAM are created. The WAM object is not a system itself, but is comprised of mainly of various systems.

The following classes are contained in every WAM object, and are instances of built-in Libbarrett systems.

lowLevelWam – This is the core of the WAM object. It represents the actual hardware of the Wam, handling communications between the software and the arm itself. This system takes joint torques as an input, converts them to motor torques, and sends the necessary CAN messages to command these torques to the Pucks. It is also responsible for receiving the motor positions from the pucks, converting this to joint positions and velocities at its output.

jtSummer – A system that adds all its inputs together and feeds the result to its output. In the case of the WAM object, this sums the input joint torques and outputs their sum as joint torques.

JpOutput and jvOutput – Virtual interfaces for joint positions and joint velocities. These systems allow you to attached their output to inputs of other systems relying on the values from the WAM arm.

toolOrientation and toolPosition – These systems comput the position and orientation of the end point of the WAM.

supervisoryController – This object is not a system. This object accepts any input which the existing/registered controllers of the WAM object can accept. It decides which controller should deal with the specified input, and passes it to that controller. It then handles the output of that controller and passes it to the jtSummer to be applied to the WAM. The input to the supervisory controller is given with the command, wam.trackReferenceSignal() call, and can be reset with the wam.idle() call.

gravityCompensator – Calculates the force of gravity on the WAM Arm and applies compensatory torques to the arm.

Other important and typical robotic control systems are part of the Libbarrett API. Some of these include:
pid_controller – Standard PID controller with configurable gains.

first_order_filter – Butterworth first order filter, configurable for high, low or bandpass filtering.

gain – Templated system allowing for multiplication based on a given gain.

low_level_wam_wrapper – Wrapped description of the lowLevelWam previously described exposing a number of useful low-level interfaces.

periodic_data_logger – System allowing for real-time data logging.

ramp – a configurable ramp system.

rate_limiter – System which limits the output of the system to match the input by a given value. The output will build to an over commanded input based on its configuration.

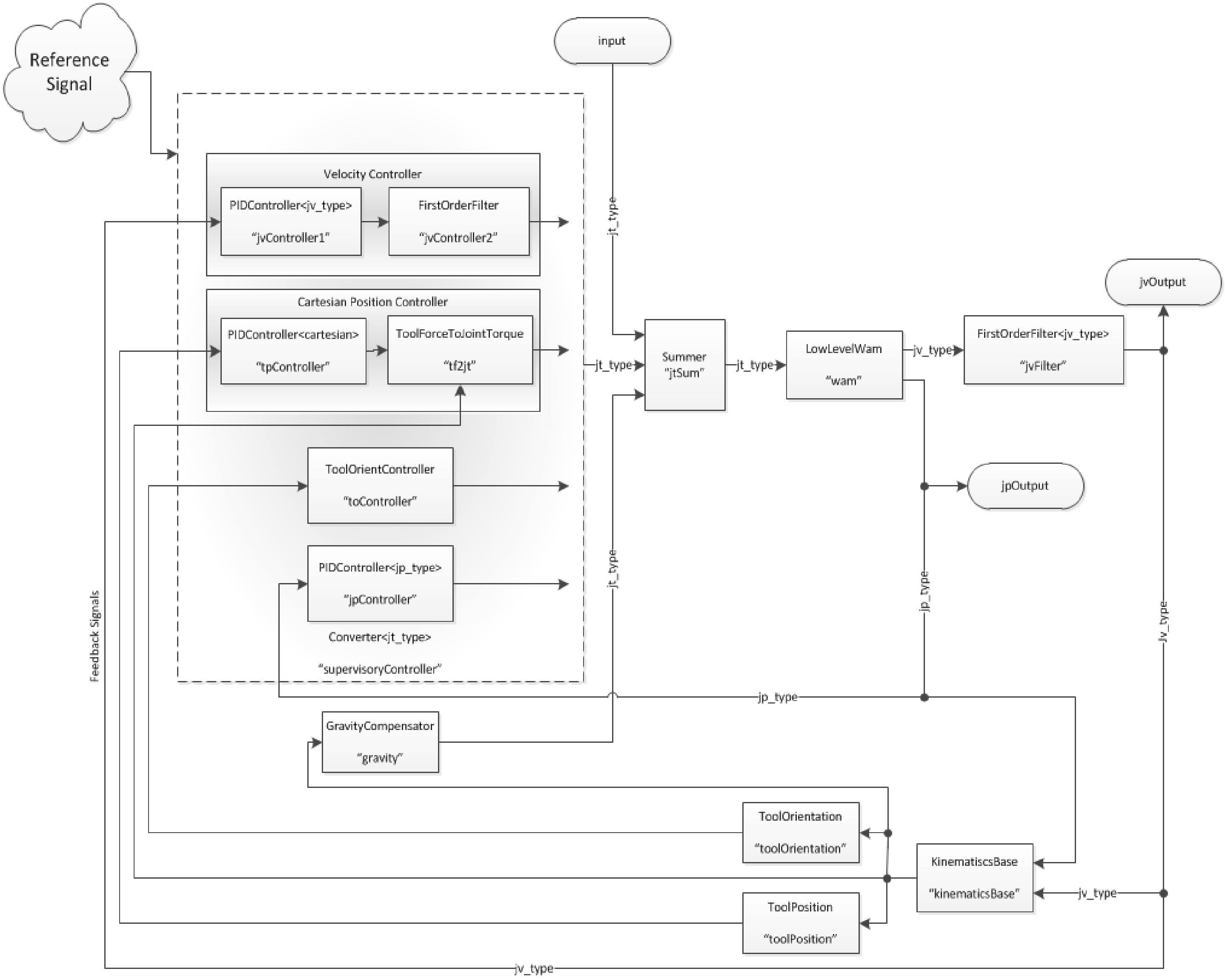
summer – System which will sum its inputs and output the result.

tool_force_to_joint_torque – System will take tool force inputs and calculate the joint torques necessary to achieve those tool forces. Based on the jacobian-transpose.

These and more can be found in the systems directory of Libbarrett.

We will cover the creation of new real-time systems later on in this document. More information can be found on page 29.

Example System Block Diagram



Libbarrett Installation

The Libbarrett library dependencies come pre-installed with any OS distributed by Barrett Technology.

Development on an external computer that will not run a WAM will require the installation of the library dependencies. This development typically takes place on an external PC and is then copied to the WAM PC/104 for execution.

These installation instructions accompany the documentation on support.barrett.com: http://support.barrett.com/wiki/Libbarrett/Installation

Please Note: While you can now develop, and compile Libbarrett programs without running our real-time operation system, you **will only** be able to run the Libbarrett software on a PC that is running our real-time operating system. Details and a description of this are located here: http://support.barrett.com/wiki/WAM/DetailedSystemOperation

We advise development of new control software on an external PC, then copying over the necessary source files to the real-time control PC (typically the WAM's internal PC/104), recompiling, and running the software.

To install Libbarrett-1.2.4, follow the instructions on our git repository (also reproduced below):
https://git.barrett.com/software/libbarrett

Test your installation by reading the documentation:
http://support.barrett.com/wiki/Libbarrett

Explore and run the examples in your home directory

```
$ # Compilation and test of example Libbarrett program.
$ cd ~/libbarrett_examples
$ cmake .
$ make ex02_hold_position
$ ./ex02_hold_position
```

Libbarrett Source Compilation and Installation (detail)

These detailed installation and compilation instructions are reproduced from our git repository at https://git.barrett.com/software/libbarrett:

Pre-requisites:

```
$ sudo apt-get install git python-dev python-argparse
$ sudo apt-get install libeigen3-dev libboost-all-dev libgsl0-dev
$ sudo apt-get install libxenomai-dev libxenomai1
$ wget http://web.barrett.com/support/WAM_Installer/libconfig-1.4.5-PATCHED.tar.gz
$ tar -xf libconfig-1.4.5-PATCHED.tar.gz
$ cd libconfig-1.4.5
$ ./configure && make && sudo make install
$ cd ../
$ rm -rf libconfig-1.4.5 libconfig-1.4.5-PATCHED.tar.gz
```

Download and install libbarrett:

```
$ cd ~/
$ git clone https://github.com/BarrettTechnology/libbarrett.git
$ cd libbarrett
$ cmake .
$ make
$ sudo make install
```

Configuration Files for the robots:

Upon installation of libbarrett, the configuration files of the robot are installed to the /etc/barrett directory. However, to give an additional flexibility of each user maintaining their own configurations for the same robot, by default, the configuration files are read from ~/.barrett directory if it exists. If not, then libbarrett reads the necessary configuration files from /etc/barrett/ directory. It is up to the user to maintain and populate the ~/.barrett directory

For Proficio support only:

In addition to the above, there are two different configuration files for the Proficio to account for its variant.. So, the right set of configuration files have to be copied either to the /etc/barrett/ or ~/.barrett folder depending on the configuration of the robot in use. In order to ease this process, there is a bash aliases that copies the corresponding configuration files to the /etc/barrett directory. As before, copying it to the ~/.barrett directory has to be done manually by the user.

To set it up for the first time after installing libbarrett,

Copy the bash_aliases to the existing aliases

```
$ cat .bash_aliases >> ~/libbarrett/.bash_aliases
```

or if no bash aliases exist then create a new one

```
$ mv libbarrett/.bash_aliases .
```

Rerun the bash script

```
$ . ~/.bashrc
```

- Before running the Proficio in a particular configuration or immediately after switching the configuration of the proficio, type either of the following based on the configuration of the robot

```
$ leftConfig
or
$ rightConfig
```

The above bash aliases copies the particular configuration of the proficio from ~/libbarrett/proficio_sandbox/configurations into /etc/barrett directory.

- Hit E-STOP and shift+idle.

- If the outer elbow of the proficio is swapped, do gravity calibration before running the examples.

Headers and shared libraries will be installed to their typical locations for your system. Configuration files will be installed to the /etc/barrett/ directory. A copy of the examples/ directory will be placed in your home folder.

The installation of Libbarrett results in the library installation in /usr/local/lib with accompanying header files in /usr/local/include. Also included in the installation are the calibration routines, bt-wam-zerocal, bt-wam-gravitycal and bt-wam-autotension installed into /usr/local/bin, now allowing the start of zero and gravity calibration from any location in the filesystem. A Libbarrett CMake-find module (barrett-config.cmake) is installed for ease of future Libbarrett inclusion and linking in /usr/local/share/barrett directory. A Libbarrett examples and sandbox folder will be placed into the users home directory. Lastly, calibration files for standard usage are installed into the /etc/barrett directory.

Creating a Simple New Libbarrett Program

Description for creation of a simple Libbarrett program. This program will use the standard main function to create the WAM object. We will then turn on gravity compensation until the user exits by pressing <SHIFT + IDLE> on the control pendant.

Create a new folder for our simple Libbarrett program.

```
$ cd ~/
$ mkdir simple_libbarrett_example
$ cd simple_libbarrett_example
```

Create our cpp source file

```
$ vim simple_example.cpp
```

```
/*** A Simple Libbarrett Example Program ***/

#include <barrett/systems.h>
#include <barrett/products/product_manager.h>
#include <barrett/standard_main_function.h>

using namespace barrett;

template<size_t DOF>
int wam_main(int argc, char** argv, ProductManager& pm,
               systems::Wam<DOF>& wam) {

      // Turn on Gravity Compensation
      wam.gravityCompensate(true);

      // Gravity compensation will wait until the user presses
      // <SHIFT + IDLE> on the control pendant.
      pm.getSafetyModule()->waitForMode(SafetyModule::IDLE);
      return 0;
}
```

Save and exit.

Create the CMakeLists.txt file necessary for compilation of our new program.

```
$ vim CMakeLists.txt
```

```
## An example CMakeLists.txt file showing usage of the CMake finder
## for linking against Libbarrett

cmake_minimum_required(VERSION 2.6)
project(simple_example)

## Libbarrett
find_package(Barrett REQUIRED)
include_directories(${BARRETT_INCLUDE_DIRS})
link_directories(${BARRETT_LIBRARY_DIRS})
add_definitions(${BARRETT_DEFINITIONS})

## Simple Example
add_executable(simple_example simple_example.cpp)
target_link_libraries(simple_example ${BARRETT_LIBRARIES})
```

Save and exit.

Compile our example program.
$ cmake .
$ make

Run the example program.
$ ./simple_example

Depending on the current state of the WAM, you will be prompted to <SHIFT + IDLE>, Place the robot in the home position, and <SHIFT + ACTIVATE> the WAM. Gravity compensation will be turn on at start up allowing the user to move the robot freely by hand. The program will prompt the user to <SHIFT + IDLE>, this is the standard way to exit a Libbarrett program successfully.

Creating a New Real-Time System

Creation of a real-time system is necessary for advanced development and usage of the WAM and Libbarrett. This example program will take the joint torques commanded to the WAM, and the current WAM joint positions as input, exposing these as a public member of the class where allowing us to print these values to the screen at a low rate. We will then command the WAM to move to a few position in space, all the while printing the torques commanded to each joint.

Create a new folder for our real-time system example program.

```
$ cd ~/
$ mkdir libbarrett_system_example
$ cd libbarrett_system_example
```

Create our cpp source file

```
$ vim libbarrett_system_example.cpp
```

```
/*** A Libbarrett Real-Time System Example Program ***/

#include <barrett/systems.h>
#include <barrett/products/product_manager.h>
#include <barrett/detail/stl_utils.h>
#include <barrett/standard_main_function.h>

using namespace barrett;

template<size_t DOF>
class ExampleSystem : public systems::System
{
  BARRETT_UNITS_TEMPLATE_TYPEDEFS(DOF);

 public:
  Input<jt_type> commandedJTIn;
  Input<jp_type> wamJPIn;
  Output<jt_type> wamJTOutput;

 protected:
  typename Output<jt_type>::Value* jtOutputValue;

 public:
  ExampleSystem(const std::string& sysName = "ExampleSystem") :
      System(sysName), commandedJTIn(this), wamJPIn(this),
      wamJTOutput(this, &jtOutputValue)
  {
  }
  virtual ~ExampleSystem()
  {
    this->mandatoryCleanUp();
  }

 protected:
  virtual void operate()
  {
    commandedJT = commandedJTIn.getValue();
    wamJP = wamJPIn.getValue();
    // If we were to do any real-time control calculations they
    // would take place here.
    jtOutputValue->setData(&commandedJT);
  }

 public:
  jt_type commandedJT;
  jp_type wamJP;

 private:
  DISALLOW_COPY_AND_ASSIGN(ExampleSystem);
};

template<size_t DOF>
int wam_main(int argc, char** argv, ProductManager& pm,
             systems::Wam<DOF>& wam)
{
  BARRETT_UNITS_TEMPLATE_TYPEDEFS(DOF);

  // Turn on Gravity Compensation
  wam.gravityCompensate();

  // Create an instance of our system
  ExampleSystem<DOF> exampleSystem;

  // Connect inputs
  systems::connect(wam.jtSum.output, exampleSystem.commandedJTIn);
  systems::connect(wam.jpOutput, exampleSystem.wamJPIn);

  printf("Press <Enter> to move to the zero position for each  joint.\n");
  detail::waitForEnter();
  jp_type zeroMove; // Defaults to all zero values

  // Normally moveTo calls are blocking, and the rest of your
  // program will not continue until the move is done. Passing
  // false will make this a non-blocking movement, allowing us to
  // monitor and print.
  wam.moveTo(zeroMove, false);

  while (!wam.moveIsDone())
  {
    if (DOF == 4)
    {
      printf("Current WAM Joint Positions- J1: %f, J2: %f, J3: %f, J4: %f\n",
             exampleSystem.wamJP[0], exampleSystem.wamJP[1],
             exampleSystem.wamJP[2], exampleSystem.wamJP[3]);
      printf("Commanded WAM Joint Torques- J1: %f, J2: %f, J3: %f, J4: %f\n",
             exampleSystem.commandedJT[0], exampleSystem.commandedJT[1],
             exampleSystem.commandedJT[2], exampleSystem.commandedJT[3]);
    }
    else
    {
      printf("Current WAM Joint Positions- J1: %f, J2: %f, J3: %f, J4: %f, J5:
%f, J6: %f, J7: %f\n",
             exampleSystem.wamJP[0], exampleSystem.wamJP[1],
             exampleSystem.wamJP[2], exampleSystem.wamJP[3],
             exampleSystem.wamJP[4], exampleSystem.wamJP[5],
             exampleSystem.wamJP[6]);
      printf("Commanded WAM Joint Torques- J1: %f, J2: %f, J3: %f, J4: %f, J5:
%f, J6: %f, J7: %f\n",
             exampleSystem.commandedJT[0], exampleSystem.commandedJT[1],
             exampleSystem.commandedJT[2], exampleSystem.commandedJT[3],
             exampleSystem.commandedJT[4], exampleSystem.commandedJT[5],
             exampleSystem.commandedJT[6]);
      }
    }
  printf("Finished Zero Move");
  printf("Press <Enter> to move WAM back to home position.\n");
  detail::waitForEnter();
  wam.moveHome(false);
  while (!wam.moveIsDone())
  {
    if (DOF == 4)
    {
      printf("Current WAM Joint Positions- J1: %f, J2: %f, J3: %f, J4: %f\n",
             exampleSystem.wamJP[0], exampleSystem.wamJP[1],
             exampleSystem.wamJP[2], exampleSystem.wamJP[3]);
      printf("Commanded WAM Joint Torques- J1: %f, J2: %f, J3: %f, J4: %f\n",
             exampleSystem.commandedJT[0], exampleSystem.commandedJT[1],
             exampleSystem.commandedJT[2], exampleSystem.commandedJT[3]);
    }
    else
    {
      printf("Current WAM Joint Positions- J1: %f, J2: %f, J3: %f, J4: %f, J5:
%f, J6: %f, J7: %f\n",
             exampleSystem.wamJP[0], exampleSystem.wamJP[1],
             exampleSystem.wamJP[2], exampleSystem.wamJP[3],
             exampleSystem.wamJP[4], exampleSystem.wamJP[5],
             exampleSystem.wamJP[6]);
      printf("Commanded WAM Joint Torques- J1: %f, J2: %f, J3: %f, J4: %f, J5:
%f, J6: %f, J7: %f\n",
             exampleSystem.commandedJT[0], exampleSystem.commandedJT[1],
             exampleSystem.commandedJT[2], exampleSystem.commandedJT[3],
             exampleSystem.commandedJT[4], exampleSystem.commandedJT[5],
             exampleSystem.commandedJT[6]);
    }
  }
  printf("Finished Home Move");
  pm.getSafetyModule()->waitForMode(SafetyModule::IDLE);
  return 0;
}
```

Save and exit.

```
## An example CMakeLists.txt file showing usage of the CMake finder
## for linking against Libbarrett

cmake_minimum_required(VERSION 2.6)
project(libbarrett_system_example)

## Libbarrett
find_package(Barrett REQUIRED)
include_directories(${BARRETT_INCLUDE_DIRS})
link_directories(${BARRETT_LIBRARY_DIRS})
add_definitions(${BARRETT_DEFINITIONS})

## Real-Time System Example
add_executable(libbarrett_system_example libbarrett_system_example.cpp)
target_link_libraries(libbarrett_system_example ${BARRETT_LIBRARIES})
```

Save and exit.

Compile our example program.
$ cmake .
$ make

Run the example program.
$ ./libbarrett_system_example

More Libbarrett Example Programs

Additional example programs were installed with libbarrett. They programs contain examples of common usage of some other real-time systems, including PID controllers and data loggers. The examples can be found in ~/libbarrett_examples.

For advanced users, some more complex program examples can be found in ~/libbarrett_sandbox.

BarrettHand Introduction

Creation of BarrettHand control software is independent of other WAM software, and therefore may be create standalone or included in typical WAM control programs.

BarrettHand Control Example

This example program will simply create an instance of the BarrettHand, and command a few simple movements.

$ cd ~/
$ mkdir simple_bhand_example
$ cd simple_bhand_example

Create our cpp source file
$ vim bhand_example .cpp

```
//*** A Simple Libbarrett BarrettHand Control Example Program ***/

#include <barrett/systems.h>
#include <barrett/products/product_manager.h>

using namespace barrett;

int main(int argc, char** argv) {
      ProductManager pm;
      if (!pm.foundHand()) {
            printf("ERROR: No Hand found on bus!\n");
            return 1;
      }
      Hand& hand = *pm.getHand();
      hand.initialize();

      hand.close();
      hand.open();
      hand.close(Hand::SPREAD);
      hand.close(Hand::GRASP);
      hand.open();
      Hand::jp_type firstPosMove;
      Hand::jp_type secondPosMove;
      firstPosMove[0] = 1.0;
      firstPosMove[1] = 2.4;
      firstPosMove[2] = 0.5;
      firstPosMove[3] = 1.57;
      secondPosMove[0] = 0.0;
      secondPosMove[1] = 1.5;
      secondPosMove[2] = 2.4;
      secondPosMove[3] = 3.14;
      hand.trapezoidalMove(firstPosMove);
      hand.trapezoidalMove(secondPosMove);

      hand.open();

      Hand::jv_type closing;
      Hand::jv_type opening;
      closing[0] = 0.25;
      closing[1] = 0.5;
      closing[2] = 1.0;
      opening[0] = -1.0;
      opening[1] = -0.25;
      opening[2] = -0.5;

      hand.velocityMove(closing,Hand::GRASP);
      btsleep(4.0); // allow the movement to happen for 4 seconds
      hand.velocityMove(opening,Hand::GRASP);
      btsleep(4.0); // allow the movement to happen for 4 seconds

      hand.open();
      return 0;
}
```

Save and exit.

Create the CMakeLists.txt file necessary for compilation of our new program.

$ vim CMakeLists.txt

```
## An example CMakeLists.txt file showing usage of the CMake finder
## for linking against Libbarrett

cmake_minimum_required(VERSION 2.6)
project(simple_bhand_example)

## Libbarrett
find_package(Barrett REQUIRED)
include_directories(${BARRETT_INCLUDE_DIRS})
link_directories(${BARRETT_LIBRARY_DIRS})
add_definitions(${BARRETT_DEFINITIONS})

## Simple Example
add_executable(bhand_example bhand_example.cpp)
target_link_libraries(bhand_example ${BARRETT_LIBRARIES})
```

Save and exit.

Compile our example program.

```
$ cmake .
$ make
```

Run the example program.

```
$ ./bhand_example
```

IDE Usage (Eclipse)

The following installation instructions are for Eclipse CDT (C/C++ Development Tooling).

Eclipse CDT allows for syntax highlighting, auto-completion, visual debugging, easy source navigation as well as other useful tools.

In any Libbarrett source or development directory you may create an Eclipse project using the command:

```
$ cmake . -G"Eclipse CDT4 - Unix Makefiles"
```

You can now import your Libbarrett project in Eclipse:
File --> Import --> General --> Existing Projects into Workspace --> Next --> Browse

Browse to your package's root directory and hit OK

Finish the import by selecting the finish button.

Your project should now be imported into Eclipse and can be viewed in the Project Explorer on the left side of the IDE.