TI Object Propagation (TIOP) using TI's TMS320C66x DSP

User Guide

December 2018

IMPORTANT NOTICE

1 Read This First

1.1 About This Manual

This document describes how to install and work with Texas Instruments' (TI) TIOP Module implemented on TI's TMS320C66x DSP. It also provides a detailed Application Programming Interface (API) reference and information on the sample application that accompanies this component.

TI's Object Propagation Module implementations are based on IVISION interface. IVISION interface is an extension of the eXpressDSP Algorithm Interface Standard (XDAIS).

1.2 Intended Audience

This document is intended for system engineers who want to integrate TI's vision and imaging algorithms with other software to build a high level vision system based on C66x DSP.

This document assumes that you are fluent in the C language, and aware of vision and image processing applications. Good knowledge of eXpressDSP Algorithm Interface Standard (XDAIS) standard will be helpful.

1.3 How to Use This Manual

This document includes the following chapters:

- **Chapter 2 - Introduction**, provides a brief introduction to the XDAIS standards. It also provides an overview of TIOP and lists its supported features.
- **Chapter 3 - Installation Overview**, describes how to install, build, and run the algorithm.
- **Chapter 4 - Sample Usage**, describes the sample usage of the algorithm.
- **Chapter 5 - API Reference**, describes the data structures and interface functions used in the algorithm.

1.4 Related Documentation From Texas Instruments

This document frequently refers TI's DSP algorithm standards called XDAIS. To obtain a copy of document related to any of these standards, visit the Texas Instruments website at www.ti.com.

1.5 Abbreviations

The following abbreviations are used in this document.

Table 1 List of Abbreviations

Abbreviation	Description
TIOP	TI Object Propagation
API	Application Programming Interface
DSP	Digital Signal Processing
EVM	Evaluation Module
IRES	Interface for Resources
QCIF	Quarter Common Intermediate Format
QVGA	Quarter Video Graphics Array
SQCIF	Sub Quarter Common Intermediate Format
VGA	Video Graphics Array
XDAIS	eXpressDSP Algorithm Interface Standard

1.6 Text Conventions

The following conventions are used in this document:

- Text inside back-quotes (‘‘) represents pseudo-code.
- Program source code, function and macro names, parameters, and command line commands are shown in a `mono-spaced` font.

1.7 Product Support

When contacting TI for support on this product, quote the product name (TIOP Module on TMS320C66x DSP) and version number. The version number of the TIOP Module is included in the Title of the Release Notes that accompanies the product release.

1.8 Trademarks

Code Composer Studio, eXpressDSP are trademarks of Texas Instruments.

2 Introduction

This chapter provides a brief introduction to XDAIS. It also provides an overview of TI's implementation of TIOP on the C66x DSP and its supported features.

2.1 Overview of XDAIS

TI's vision analytics applications are based on IVISION interface. IVISION is an extension of the eXpressDSP Algorithm Interface Standard (XDAIS). Please refer documents related to XDAIS for further details.

2.1.1 XDAIS Overview

An eXpressDSP-compliant algorithm is a module that implements the abstract interface IALG. The IALG API takes the memory management function away from the algorithm and places it in the hosting framework. Thus, an interaction occurs between the algorithm and the framework. This interaction allows the client application to allocate memory for the algorithm and also share memory between algorithms. It also allows the memory to be moved around while an algorithm is operating in the system. In order to facilitate these functionalities, the IALG interface defines the following APIs:

- `algAlloc()`
- `algInit()`
- `algActivate()`
- `algDeactivate()`
- `algFree()`

The `algAlloc()` API allows the algorithm to communicate its memory requirements to the client application. The `algInit()` API allows the algorithm to initialize the memory allocated by the client application. The `algFree()` API allows the algorithm to communicate the memory to be freed when an instance is no longer required.

Once an algorithm instance object is created, it can be used to process data in real-time. The `algActivate()` API provides a notification to the algorithm instance that one or more algorithm processing methods is about to be run zero or more times in succession. After the processing methods have been run, the client application calls the `algDeactivate()` API prior to reusing any of the instance's scratch memory.

The IALG interface also defines three more optional APIs `algControl()`, `algNumAlloc()`, and `algMoved()`. For more details on these APIs, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

2.2 Overview of TIOP

The object detection algorithms typically work on image basis and detect objects without using temporal information across images. In a real-life use case of automotive, the camera captures a sequence of images which are temporally consistent. The purpose of this object propagation module is to improve the detection quality by applying temporal knowledge. It accepts a list of detected objects from any detection algorithm and applies propagation strategy.

Any object detection algorithm (for example Deep learning base jdetnet network) need to become very compute intensive in order to achieve very high objective and visual quality. Beyond a certain point even to gain marginal improvement in quality, the additional complexity becomes very high when the temporal information is not exercised. Whereas with using temporal information we can achieve better quality with very marginal additional complexity. TI Object propagation module is created on this principle. So an object detection algorithm can be little relaxed in detecting even moderately confident objects and can feed to object propagation module to filter out all potential false detections.

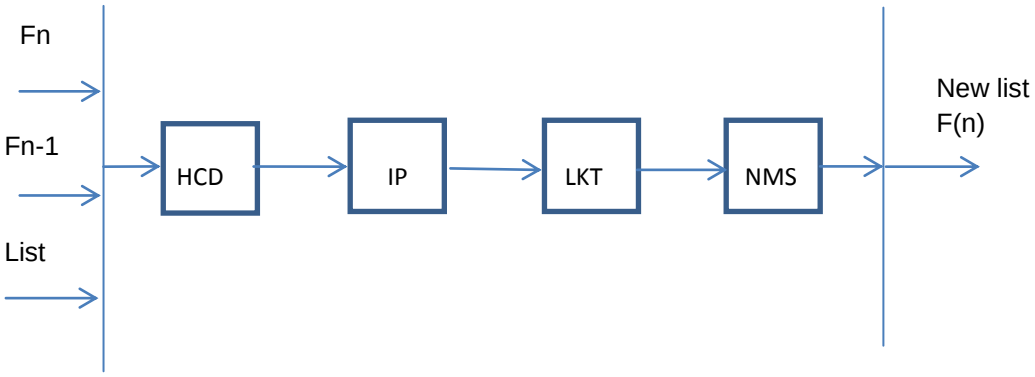

Figure 1 Fundamental blocks of TIOP

2.3 Supported Services and Features

This user guide accompanies TI's implementation of TIOP Algorithm on the TI's C66x DSP.

This version of the TIOP has the following supported features of the standard:

- Supports tracking of objects from previous frame to current frame
- Supports maximum of 5 number of Levels for LK tracker and supports this parameter as configurable
- Supports user controlled parameters (threshold and scaling factor) for Harries corner detection
- Supports external computed image pyramidal data buffers as input

- Independent of any operating system

3 Installation Overview

This chapter provides a brief description on the system requirements and instructions for installing TIOP module. It also provides information on building and running the sample test application.

3.1 System Requirements

This section describes the hardware and software requirements for the normal functioning of the algorithm component.

3.1.1 Hardware

This algorithm has been built and tested TI's C66x DSP on TDA2x platform. The algorithm shall work on any future TDA platforms hosting C66x DSP.

3.1.2 Software

The following are the software requirements for the stand alone functioning of the TIOP module:

- **Development Environment:** This project is developed using TI's Code Generation Tool 7.4.2. Other required tools used in development are mentioned in section 3.3
- The project are built using g-make (GNU Make version 3.81). GNU tools comes along with CCS installation.

3.2 Installing the Component

The algorithm component is released as install executable. Following sub sections provided details on installation along with directory structure.

3.2.1 Installing the compressed archive

The algorithm component is released as a compressed archive. To install the algorithm, extract the contents of the zip file onto your local hard disk. The zip file extraction creates a top-level directory called ti_object_propagation.Folder structure of this top level directory is shown in below figure.

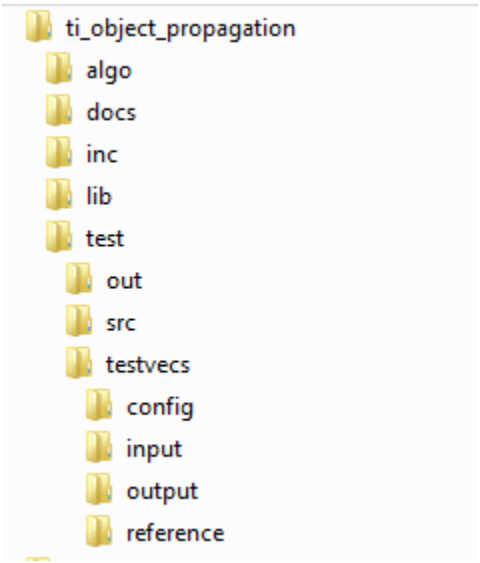

Figure 2 Component Directory Structure In case of Object Release

Table 2 Component Directories in case of Object release

Sub-Directory	Description
\ti_object_Propagation	TI Object propagation module for C66x DSP
\ti_object_Propagation \docs	User guide and Datasheet for TI Object propagation module
\ti_object_Propagation \inc	Contains iop_ti.h interface file
\ti_object_Propagation \lib	Contains Object propagation module library
\ti_object_Propagation \test	Contains standalone test application source files
\ti_object_Propagation \test\out	Contains test application .out executable
\ti_object_Propagation \test\src	Contains test application source files
\ti_object_Propagation \test\testvecs	Contains config, input, output, reference test vectors
\ti_object_Propagation \test\testvecs\config	Contain config file to set various parameters exposed by TI Object propagation module
\ti_object_Propagation \test\testvecs\input	Contains sample input feature vector .bin file

Sub-Directory	Description
\ti_object_Propagation \test\testvecs\output	Contains output file
\modules\ti_dl \test\testvecs\reference	Contains reference file

3.3 Building Sample Test Application

This TIOP library has been accompanied by a sample test application. To run the sample test application XDAIS tools are required.

This version of the TIOP library has been validated with XDAIS tools containing IVISION interface version. Other required components (for test application building) version details are provided below.

- Refer to release notes for dependent component and their versions

3.3.1 Installing XDAIS tools (XDAIS)

XDAIS version 7.22 can be downloaded from the following website:

http://downloads.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/xdais/

Extract the XDAIS zip file to the same location where Code Composer Studio has been installed. For example:

C:\CCStudio5.0

Set a system environment variable named "XDAIS_PATH" pointing to <install directory>\<xdais_directory>

3.3.2 Installing C66x VLIB

Install C66x VLIB version from the link

http://software-dl.ti.com/libs/vlib/latest/index_FDS.html
After installing VLIB, set the environment variable to "VLIB_PATH" to the installed directory like <install directory>

3.3.3 Installing Code Generation Tools

Install Code generation Tools version 7.4.2 from the link

https://www-a.ti.com/downloads/sds_support/TICodegenerationTools/download.htm

After installing the CG tools, set the environment variable to "DSP_TOOLS" to the installed directory like <install directory>\<cgtools_directory>

3.3.4 Building the Test Application Executable through GMAKE

The sample test application that accompanies TIOP module will run in TI's Code Composer Studio development environment. To build and run the sample test application through gmake, follow these steps:

1) Verify that you have installed code generation tools version 7.4.2.

2) Verify that you have installed XDAIS tools version 7.22.00.03

3) Verify that appropriate environment variables have been set as discussed in this above sections.

4) Build the sample test application project by gmake

    a) modules\ ti_object_propagation\test> gmake clean

    b) modules\ ti_object_propagation\test> gmake all

5) The above step creates an executable file, dsp_test_op_algo.out in the modules\ti_object_propagation\test\out sub-directory.

6) Open CCS with TDA2x platform selected configuration file. Select Target > Load Program on C66x DSP, browse to the modules\ ti_object_propagation\test\out sub-directory, select the executable created in step 5, and load it into Code Composer Studio in preparation for execution.

7) Select Target > Run on C66x DSP window to execute the sample test application.

8) Sample test application takes the input files stored in the \test\testvecs\input sub-directory, runs the module.

9) The reference files stored in the \test\testvecs\reference sub-directory can be used to verify that the TIOP is functioning as expected.

10) On successful completion, the test application displays the information for each feature frame and writes the information regarding the detected objects in the \test\testvecs\output sub-directory.

11) User should compare with the reference provided in \test\testvecs\reference directory. Both the content should be same to conclude successful execution.

3.4 Configuration File

This algorithm is shipped along with:

- Algorithm configuration file (tiop_config.cfg) – specifies the configuration parameters used by the test application to configure the Algorithm.

3.4.1 Test Application Configuration File

The algorithm configuration file, tiop_config.cfg contains the configuration parameters required for the algorithm. The tiop_config.cfg file is available in the \test\testvecs\config sub-directory.

A sample tiop_config.cfg file is as shown.

```
#######################################################

# Configuration Parameters For object propagation

#######################################################

# input data frame in Y(Luma) format

inDataFrame         = "..\..\test\testvecs\input\VIRB0003_4400sfr_5fr.y"



# List of detections in the input data file

inDataList          =
       "..\..\test\testvecs\input\VIRB0003_4400sfr_5fr_detectInList.txt"



# Externally calculated Image pyramidal buffer is provided to TIOP module if this is
set to 1

extImagePyramidal = 0



# Externally calculated image pyramidal buffers, applicable only when
extImagePyramidal is set to 1

inImagePyramidal       =     "NA"



# Externally calculated optical flow buffer is provided to TIOP module if this is set
to 1(Not supported in this release)

extOpticalFlow = 0



# Externally calculated dense optical flow buffer, applicable only when extOpticalFlow
is set

inOpticalFlow       = "NA"
```

```
# New list of detections for the input data file

outputFile          =
       "..\..\test\testvecs\output\VIRB0003_4400sfr_5fr_detectOutList.bin"


# Max Width of the input frame

inMaxWidth        =   1280


# Max Height of the input frame

inMaxHeight     =     720


# Actual width of the input frame

srcImageWidth     =   1280


# Actual height of the input frame

srcImageHeight  =     720

# Pitch of the input frame

srcImagePitch     =   1280


# Number of detected objects in the input

numInDetections  =  200


# Number of levels for LK tracker

numLevels             =      4


# Max iterations for LK tracker

maxItersLK          = 10


# Threshold for LK tracker error
```

```
lkErrThresh    =      500


# number of classes in input detected objects

numClasses     =      21


# Thresholed for Harries corner

nmsThresh              =      286870912


# Scaling Factor for Harries corner

harrisScoreScalingFactor     =      1310


# Confidence score for good objects

confScoreGoodObjs      =      0.4


# Confidence score for Moderate objects

confScoreModObjs       =      0.12


# Threshold for overlap NMS

maxOverlapThresh       =      0.4


# Threshold for max age of the propagted object

maxAgeThresh   =      30
```

If you specify additional fields in the tiop_config.cfg file, ensure that you modify the test application appropriately to handle these fields.

3.5 Host emulation build for source package

The source release of TIOP module can be built in host emulation mode. This option speeds up development and validation time by running the platform code on x86/x64 PC.

3.5.1 Installing Visual Studio

Building host emulation for Circular Light Recognition module requires Microsoft Visual Studio 11.0 (2012) which can be downloaded from below link.

http://www.microsoft.com/en-in/download/details.aspx?id=34673

3.5.2 Building source in host emulation

After installing the required components, navigate to TIOP install path and run `vcvarsall.bat` to setup the required environment variables

```
{ti_object_propagation_install_path} > {…\Microsoft Visual Studio
11.0\VC\vcvarsall.bat}
```

Once the environment variables are setup build the TIOP source in host emulation mode

```
{ ti_object_propagation_install_path } > gmake all
TARGET_BUILD=debug TARGET_PLATFORM=PC
```

This will build the host emulation executable under the path

```
{ ti_object_propagation_install_path}\test\out\
dsp_test_op_algo.out.exe
```

3.5.3 Running host emulation executable

Launch Microsoft Visual Studio 11.0 and open file `dsp_test_op_algo.out.exe`

This will load the host emulation program which can be used for development and validation purpose.

3.6 Uninstalling the Component

To uninstall the component, delete the algorithm directory from your hard disk.

4 Sample Usage

This chapter provides a detailed description of the sample test application that accompanies this TIOP component.

4.1 Overview of the Test Application

The test application exercises the `IVISION` and extended class of the TIOP library. The source files for this application are available in the \test\src sub-directories.

Test Application	XDAIS – IVISION interface	DSP Apps
Algorithm instance creation and initialization	algNumAlloc() algAlloc() algInit()	
Process Call	control() process() control()	
Algorithm instance deletion	algNumAlloc() algFree()	

Table 3 Test Application Sample Implementation

The test application is divided into four logical blocks:

- Parameter setup
- Algorithm instance creation and initialization
- Process call
- Algorithm instance deletion

4.2 Parameter Setup

Each algorithm component requires various configuration parameters to be set at initialization. For example, TIOP requires parameters such as image height, image width, and so on. The test application obtains the required parameters from the Algorithm configuration files.

In this logical block, the test application does the following:

1) Opens the configuration file, listed in config_list.txt and reads the various configuration parameters required for the algorithm.

For more details on the configuration files, see Section 3.4.

2) Sets the `TIOP_CreateParams` structure based on the values it reads from the configuration file.

3) Does the algorithm instance creation and other handshake via. control methods

4) For each frame reads the feature planes into the application input buffer and makes a process call

5) For each frame dumps out the detected points along with meta data to specified output file.

4.3 Algorithm Instance Creation and Initialization

In this logical block, the test application accepts the various initialization parameters and returns an algorithm instance pointer. The following APIs implemented by the algorithm are called in sequence by `ALG_create()`:

1) `algNumAlloc()` - To query the algorithm about the number of memory records it requires.

2) `algAlloc()` - To query the algorithm about the memory requirement to be filled in the memory records.

3) `algInit()` - To initialize the algorithm with the memory structures provided by the application.

A sample implementation of the create function that calls `algNumAlloc()`, `algAlloc()`, and `algInit()` in sequence is provided in the `ALG_create()` function implemented in the alg_create.c file.

4.4 Process Call

After algorithm instance creation and initialization, the test application does the following:

1) Sets the dynamic parameters (if they change during run-time) by calling the `control()` function with the `IALG_SETPARAMS` command.

2) Sets the input and output buffer descriptors required for the `process()` function call. The input and output buffer descriptors are obtained by calling the `control()` function with the `IALG_GETBUFINFO` command.

3) Calls the `process()` function to detect propagated objects in the provided feature plane. The inputs to the process function are input and output buffer

descriptors, pointer to the `IVISION_InArgs` and `IVISION_OutArgs` structures.

4) When the `process()` function is called, the software triggers the start of algorithm.

The `control()` and `process()` functions should be called only within the scope of the `algActivate()` and `algDeactivate()` XDAIS functions, which activate and deactivate the algorithm instance respectively. If the same algorithm is in-use between two process/control function calls, calling these functions can be avoided. Once an algorithm is activated, there can be any ordering of `control()` and `process()` functions. The following APIs are called in sequence:

1) `algActivate() – To activate the algorithm instance.`

2) `control()` (optional) - To query the algorithm on status or setting of dynamic parameters and so on, using the eight control commands.

3) `process()` - To call the Algorithm with appropriate input/output buffer and arguments information.

4) `control()` (optional) - To query the algorithm on status or setting of dynamic parameters and so on, using the eight available control commands.

5) `algDeactivate()` - To deactivate the algorithm instance.

The do-while loop encapsulates frame level `process()` call and updates the input buffer pointer every time before the next call. The do-while loop breaks off either when an error condition occurs or when the input buffer exhausts.

If the algorithm uses any resources through RMAN, then user must activate the resource after the algorithm is activated and deactivate the resource before algorithm deactivation.

4.5 Algorithm Instance Deletion

Once `process` is complete, the test application must release the resources granted by the IRES resource Manager interface if any and delete the current algorithm instance. The following APIs are called in sequence:

1) `algNumAlloc()` - To query the algorithm about the number of memory records it used.

2) `algFree()` - To query the algorithm to get the memory record information.

A sample implementation of the delete function that calls `algNumAlloc()` and `algFree()` in sequence is provided in the `ALG_delete()` function implemented in the alg_create.c file.

4.6 Frame Buffer Management

4.6.1 Input and Output Frame Buffer

The algorithm has input buffers that stores frames until they are processed. These buffers at the input level are associated with a bufferId mentioned in input buffer descriptor. The output buffers are similarly associated with bufferId mentioned in the output buffer descriptor. The IDs are required to track the buffers that have been

processed or locked. The algorithm uses this ID, at the end of the process call, to inform back to application whether it is a free buffer or not. Any buffer given to the algorithm should be considered locked by the algorithm, unless the buffer is returned to the application through `IVISION_OutArgs->inFreeBufID[] and IVISION_OutArgs->outFreeBufID[]`. **At Present TIOP locks one Input Buffer in each process call.**

For example,

Process Call #	1	2	3	4	5
bufferID (input)	1	2	3	4	5
bufferID (output)	1	2	3	4	5
inFreeBufID	-	1	2	3	4,5
outFreeBufID	1	2	3	4	5

5 API Reference

This chapter provides a detailed description of the data structures and interfaces functions used by TIOP .

5.1.1 IVISION_Params

Description

This structure defines the basic creation parameters for all vision applications.

Fields

Field	Data Type	Input/ Output	Description
algParams	IALG_Params	Input	IALG Params
cacheWriteBack	ivisionCacheWriteBack	Input	Function pointer for cache write back for cached based system. If the system is not using cache fordata memory then the pointer can be filled with NULL. If the algorithm recives a input buffer with IVISION_AccessMode as IVISION_ACCESSMODE_CPU and the ivisionCacheWriteBack as NULL then the algorithm will return with error

5.1.2 IVISION_Point

Description

This structure defines a 2-dimensional point

Fields

Field	Data Type	Input/ Output	Description
X	XDAS_Int32	Input	X (horizontal direction offset)
y	XDAS_Int32	Input	Y (vertical direction offset)

5.1.3 IVISION_Rect

Description

This structure defines a rectangle

Fields

Field	Data Type	Input/ Output	Description
topLeft	XDAS_Int32	Input	Top left co-ordinate of rectangle
width	XDAS_Int32	Input	Width of the rectangle
height	XDAS_Int32	Input	Height of the rectangle

5.1.4 IVISION_BufPlanes

Description

This structure defines a generic plane descriptor

Fields

Field	Data Type	Input/ Output	Description
buf	Void*	Input	Pointer to the buffer
width	XDAS_UInt32	Input	Width of the buffer (in bytes), This field can be viewed as pitch while processing a ROI in the buffer
height	XDAS_UInt32	Input	Height of the buffer (in lines)
frameROI	IVISION_Rect	Input	Region of the intererst for the current frame to be processed in the buffer. Dimensions need to be a multiple of internal block dimenstions. Refer application specific details for block dimensions supported for the algorithm. This needs to be filled even if bit-0 of IVISION_InArgs::subFrameInfo is set to 1
subFrameROI	IVISION_Rect	Input	Region of the intererst for the current sub frame to be processed in the buffer. Dimensions need to be a multiple of internal block dimenstions. Refer application specific details for block dimensions supported for the application. This needs to be filled only if bit-0 of IVISION_InArgs::subFrameInfo is set to 1

Field	Data Type	Input/ Output	Description
freeSubFrameROI	IVISION_Rect	Input	This ROI is portion of subFrameROI that can be freed after current slice process call. This field will be filled by the algorithm at end of each slice processing for all the input buffers (for all the output buffers this field needs to be ignored). This will be filled only if bit-0 of IVISION_InArgs::subFrameInfois set to 1
planeType	XDAS_Int32	Input	Content of the buffer - for example Y component of NV12
accessMask	XDAS_Int32	Input	Indicates how the buffer was filled by the producer, It is IVISION_ACCESSMODE_HWA or IVISION_ACCESSMODE_CPU

5.1.5 IVISION_BufDesc

Description

This structure defines the iVISION buffer descriptor

Fields

Field	Data Type	Input/ Output	Description
numPlanes	XDAS_Int32	Input	Number of image planes
bufPlanes[IVISION_MAX_NUM_PLANES]	IVISION_BufPlanes	Input	Description of each plane
formatType	XDAS_UInt32	Input	Format of the buffer data
bufferId	XDAS_Int32	Input	Identifier to be attached with the input frames to be processed. It is useful when algorithm requires buffering for input buffers. Zero is not supported buffer id and a reserved value
Reserved[2]	XDAS_UInt32	Input	Reserved for later use

5.1.6 IVISION_BufDescList

Description

This structure defines the iVISION buffer descriptor list. IVISION_InBufs and IVISION_OutBufs is of the same type

Fields

Field	Data Type	Input/ Output	Description
Size	XDAS_UInt32	Input	Size of the structure
numBufs	XDAS_UInt32	Input	Number of elements of type IVISION_BufDesc in the list
bufDesc	IVISION_BufDesc **	Input	Pointer to the list of buffer descriptor

5.1.7 IVISION_InArgs

Description

This structure defines the iVISION input arguments

Fields

Field	Data Type	Input/ Output	Description
Size	XDAS_UInt32	Input	Size of the structure
subFrameInfo	XDAS_UInt32	Input	bit0 - Sub frame processing enable (1) or disabled (0) bit1 - First subframe of the picture (0/1) bit 2 - Last subframe of the picture (0/1) bit 3 to 31 – reserved

5.1.8 IVISION_OutArgs

Description

This structure defines the iVISION output arguments

Fields

Field	Data Type	Input/ Output	Description
Size	XDAS_UInt32	Input	Size of the structure

Field	Data Type	Input/ Output	Description
`inFreeBufIDs[IVISION_MAX_NUM_FREE_BUFFERS]`	`XDAS_UInt32`	Input	Array of bufferId's corresponding to the input buffers that have been unlocked in the Current process call. The input buffers released by the algorithm are indicated by their non-zero ID (previously provided via IVISION_BufDesc#bufferId A value of zero (0) indicates an invalid ID. The first zero entry in array will indicate end of valid inFreeBufIDs within the array hence the application can stop searching the array when it encounters the first zero entry. If no input buffer was unlocked in the process call, inFreeBufIDs[0] will have a value of zero.
`outFreeBufIDs [IVISION_MAX_NUM_FREE_BUFFERS]`	`XDAS_UInt32`	Input	Array of bufferId's corresponding to the Output buffers that have been unlocked in the Current process call. The output buffers released by the algorithm are indicated by their non-zero ID (previously provided via IVISION_BufDesc#bufferId A value of zero (0) indicates an invalid ID. The first zero entry in array will indicate end of valid inFreeBufIDs within the array hence the application can stop searching the array when it encounters the first zero entry. If no output buffer was unlocked in the process call, inFreeBufIDs[0] will have a value of zero.
`reserved[2]`	`XDAS_UInt32`		Reserved for future usage

5.1.9 TIOP Data Structures

This section includes the following TIOP specific extended data structures:

- `TIOP_CreateParams`
- `TIOP_InArgs`

- `TIOP_outArgs`
- `TIOP_objectDescriptor`
- `TIOP_inputList`
- `TIOP_outputList`

5.1.9.1 TIOP_CreateParams

‖ Description

This structure defines the create-time input arguments for TIOP Algorithm instance object.

‖ Fields

Field	Data Type	Input/ Output	Description
`visionParams`	`IVISION_Params`	Input	See `IVISION_Params` data structure for details
`inMaxWidth`	`int32_t`	Input	Maximum image Width of the input frame to create alg handle
`inMaxHeight`	`int32_t`	Input	Maximum image Width of the input frame to create alg handle
`numLevels`	`int32_t`	Input	Number of Levels for LK Tracker
`maxItersLK`	`int32_t`	Input	Number of iterations for LK Tracker
`harrisScoreScalingFactor`	`int32_t`	Input	Harris Score Factor (k) in 0.15 format
`nmsThresh`	`int32_t`	Input	Threshold for Harris Score for NMS
`maxAgeThresh`	`int32_t`	Input	Threshold for max age of the propagated object
`lkErrThresh`	`int32_t`	Input	Threshold for LK tracker error
`numClasses`	`int32_t`	Input	number of classes in input detected objects
`numInDetections`	`int32_t`	Input	Total number of detected objects in input frame
`extImagePyramidal`	`int32_t`	Input	Flag to indicate externally calculated Image pyramidal buffers are provided to TIOP module
`extOpticalFlow`	`int32_t`	Input	Flag to indicate externally calculated dense optical flow buffer is provided to TIOP module
`l1MemSize`	`int32_t`	Input	Indicates the available l1MemSize

Field	Data Type	Input/ Output	Description
			for TIOP
l2MemSize	int32_t	Input	Indicates the available l2MemSize (L2SRAM) for TIOP
confScoreGoodObjs	float	Input	Confidence score for good objects
confScoreModObjs	float	Input	Confidence score for Moderate objects
maxOverlapThresh	float	Input	Threshold for overlap NMS

5.1.9.2 TIOP_InArgs

‖ Description

This structure contains all the parameters which are given as input to TIOP algorithm at frame level

‖ Fields

Field	Data Type	Input/ Output	Description
iVisionInArgs	IVISION_InArgs	Input	See IVISION_InArgs data structure for details.
numInBufs	int32_t	Input	Total number of input buffers to TIOP module
numInObjects	int32_t	Input	Number of detected objects in the input frame

5.1.9.3 TIOP_objectDescriptor

‖ Description

This structure contains detected object properties such as location

(xmin, ymin),(xmax, ymax), confidence (score) Fields

Field	Data Type	Input/ Output	Description
objId	float	Input, Output	Unique value to identify the number of detected object, this value is >= 0 for valid objects, -1 means not

Field	Data Type	Input/ Output	Description
			valid objects and rest of the objects after "-1" in the list are not valid
objClass	float	Input, Output	Indicates the class of object detected
objScore	Float	Input, Output	Indicates the score for the detected object
xMin	Float	Input, Output	Location minimum (top left) of the detected object along X direction. In the inBuf and outBuf, these values are from 0 to 1 as they are divided by width
yMin	Float	Input, Output	Location minimum (top left) of the detected object along Y direction. In the inBuf and outBuf, these values are from 0 to 1 as they are divided by height.
xMax	float	Input, Output	Location maximum (bottom right) of the detected object along X direction. In the inBuf and outBuf, these values are from 0 to 1 as they are divided by width
yMax	float	Input, Output	Location maximum (bottom right) of the detected object along Y direction. In the inBuf and outBuf, these values are from 0 to 1 as they are divided by height.

5.1.9.4 TIOP_OutArgs

‖ Description

This structure contains all the parameters which are given as output by the algorithm.

‖ Fields

Field	Data Type	Input/ Output	Description
iVisionOutArgs	IVISION_OutArgs	Output	See IVISION_OutArgs data structure for details.
numOutBufs	int32_t	Output	Number of output buffers from the TIOP module.
numOutObjects	int32_t	Output	Number of new list of detected objects by TIOP module.

5.1.9.5 TIOP_ inputList

‖ Description

This This is input buffer structure given to object propagation module along with input frame buffer. It contains the number of objects detected and TIOP_MAX_DETECT_OBJECTS instances of TIOP_objectDescriptor structure. The number of valid descriptors is governed by numInObjects variable.

‖ Fields

Field	Data Type	Input/ Output	Description
numInObjects	int32_t	Input	Number of detected objects in the input frame
inObjDesc[TIOP_MAX_DETECT_OBJECTS]	TIOP_objectDescriptor	Output	See TIOP_objectDescriptor details in section 5.1.9.3

5.1.9.6 TIOP_outputList

‖ Description

This is output buffer structure given out by object propagation module. It contains the number of new list of objects and TIOP_MAX_DETECT_OBJECTS instances of TIOP_objectDescriptor structure. The number of valid descriptors is governed by numOutObjects variable.

‖ Fields

Field	Data Type	Input/ Output	Description
numOutObjects	Int32_t	Output	Number of new list of objects by object propagation module
outObjDesc[TIOP_MAX_DETECT_OBJECTS]	TIOP_objectDescriptor	Output	List of detected object descriptor for output. Max size TIOP_MAX_DETECT_OBJECTS

5.2 Input, Output Buffer format

Input will be provided in 2 buffers plus in 4 more buffers (image pyramidal data will be provided if calculated externally indicated using extImagePyramidal parameter), first buffer is input frame data in Y format in which the object propagation will be done, second buffer is the detected objects in the input frame, and third, fourth, fifth and sixth input buffers (numLevels -1) is the external image pyramidal buffers for the corresponding input frame (this buffer is provided only when this “extImagePyramidal” is set to 1). These input buffer format are, inBufs::bufDesc[0]:: bufPlanes[TIOP_IN_BUFDESC_INPUT_FRAME] :: width and height are same as the

input video frame width and height, the next input buffer inBufs::bufDesc[0]:: bufPlanes[TIOP_IN_BUFDESC_DETECTION_LIST] :: width will be the TIOP_MAX_DETECT_OBJECTS times the size of each object, the third to sixth buffers inBufs::bufDesc[0]:: bufPlanes[TIOP_IN_BUFDESC_IPBUF_LEVEL1] :: width will be externally calculated image pyramidal buffer size and height is 1, when this buffer is provided the TIOP module skips calculating the image pyramidal and uses this externally provided image pyramidal buffers. Output buffer is the set of new detected objects of ' TIOP_objectDescriptor' size placed in linear memory. Since number of output buffer is not known at the time of process call, hence out buf should be allocated assuming maximum possible number of output detections. So the Maximum number of output detections is TIOP_MAX_DETECT_OBJECTS. Hence user should set outBufs :: bufDesc [TIOP_OUT_BUFDESC_OBJECT_LIST] :: bufPlanes[0] ::width equal to TIOP_MAX_DETECT_OBJECTS * sizeof(TIOP_objectDescriptor) was set at the time of create.

5.3 Interface Functions

This section describes the Application Programming Interfaces (APIs) used by TIOP . The APIs are logically grouped into the following categories:

- **Creation** – `algNumAlloc()`, `algAlloc()`
- **Initialization** – `algInit()`
- **Control** – `control()`
- **Data processing** – `algActivate()`, `process()`, `algDeactivate()`
- **Termination** – `algFree()`

You must call these APIs in the following sequence:

1) `algNumAlloc()`
2) `algAlloc()`
3) `algInit()`
4) `algActivate()`
5) `process()`
6) `algDeactivate()`
7) `algFree()`

`control()` can be called any time after calling the `algInit()` API.

`algNumAlloc(), algAlloc(), algInit(), algActivate(), algDeactivate(), and algFree() are standard XDAIS APIs. This document includes only a brief description for the standard XDAIS APIs.` For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

5.4 Creation APIs

Creation APIs are used to create an instance of the component. The term creation could mean allocating system resources, typically memory.

‖ Name

`algNumAlloc()` – determine the number of buffers that an algorithm requires

‖ Synopsis

XDAS_Int32 algNumAlloc(Void);

‖ Arguments

Void

‖ Return Value

XDAS_Int32; /* number of buffers required */

‖ Description

`algNumAlloc()` returns the number of buffers that the `algAlloc()` method requires. This operation allows you to allocate sufficient space to call the `algAlloc()` method.

`algNumAlloc()` may be called at any time and can be called repeatedly without any side effects. It always returns the same result. The `algNumAlloc()` API is optional.

For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

‖ See Also

algAlloc()

Name

`algAlloc()` – determine the attributes of all buffers that an algorithm requires

‖ Synopsis

XDAS_Int32 algAlloc(const IALG_Params *params, IALG_Fxns **parentFxns, IALG_MemRec memTab[]);

‖ Arguments

IALG_Params *params; /* algorithm specific attributes */

IALG_Fxns **parentFxns;/* output parent algorithm functions */

IALG_MemRec memTab[]; /* output array of memory records */

‖ Return Value

XDAS_Int32 /* number of buffers required */

‖ Description

`algAlloc()` returns a table of memory records that describe the size, alignment, type, and memory space of all buffers required by an algorithm. If successful, this function returns a positive non-zero value indicating the number of records initialized.

The first argument to `algAlloc()` is a pointer to a structure that defines the creation parameters. This pointer may be `NULL`; however, in this case, `algAlloc()` must assume default creation parameters and must not fail.

The second argument to `algAlloc()` is an output parameter. `algAlloc()` may return a pointer to its parent's IALG functions. If an algorithm does not require a parent object to be created, this pointer must be set to `NULL`.

The third argument is a pointer to a memory space of size `nbufs * sizeof(IALG_MemRec)` where, `nbufs` is the number of buffers returned by `algNumAlloc()` and `IALG_MemRec` is the buffer-descriptor structure defined in ialg.h.

After calling this function, `memTab[]` is filled up with the memory requirements of an algorithm.

For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

‖ See Also

algNumAlloc()
algFree()

5.5 Initialization API

Initialization API is used to initialize an instance of the algorithm. The initialization parameters are defined in the `IVISION_Params` structure (see section 5.1.1 for details).

‖ Name

`algInit()` – initialize an algorithm instance

‖ Synopsis

XDAS_Int32 algInit(IALG_Handle handle, IALG_MemRec memTab[], IALG_Handle parent, IALG_Params *params);

‖ Arguments

IALG_Handle handle; /* algorithm instance handle*/

IALG_memRec memTab[]; /* array of allocated buffers */

IALG_Handle parent; /* handle to the parent instance */

IALG_Params *params; /*algorithm init parameters */

‖ Return Value

IALG_EOK; /* status indicating success */

IALG_EFAIL; /* status indicating failure */

‖ Description

`algInit()` performs all initialization necessary to complete the run time creation of an algorithm instance object. After a successful return `from` algInit(), the instance object is ready to be used to process data.

The first argument to `algInit()` is a handle to an algorithm instance. This value is initialized to the base field of `memTab[0]`.

The second argument is a table of memory records that describe the base address, size, alignment, type, and memory space of all buffers allocated for an algorithm instance. The number of initialized records is identical to the number returned by a prior call to `algAlloc()`.

The third argument is a handle to the parent instance object. If there is no parent object, this parameter must be set to `NULL`.

The last argument is a pointer to a structure that defines the algorithm initialization parameters.

For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

Since there is no mechanism to return extended error code for unsupported parameters, this version of algorithm returns `IALG_EOK` even if some parameter unsupported is set. But subsequence control/process call it returns the detailed error code

‖ See Also

`algAlloc(),`
`algMoved()`

5.6 Control API

Control API is used for controlling the functioning of the algorithm instance during run-time. This is done by changing the status of the controllable parameters of the algorithm during run-time. These controllable parameters are defined in the `IALG_Cmd` data structure.

‖ Name

`control()` – change run time parameters and query the status

‖ Synopsis

XDAS_Int32 (*control) (IVISION_Handle handle, IALG_Cmd id, IALG_Params *inParams, IALG_Params *outParams);

‖ Arguments

IVISION_Handle handle; /* algorithm instance handle */

IALG_Cmd id; /* algorithm specific control commands*/

IALG_Params *inParams /* algorithm input parameters */

IALG_Params *outParams /* algorithm output parameters */

‖ Return Value

IALG_EOK; /* status indicating success */

IALG_EFAIL; /* status indicating failure */

‖ Description

This function changes the run time parameters of an algorithm instance and queries the algorithm's `status. control()` must only be called after a successful `call to algInit()` and must never be called after a call `to algFree()`.

The first argument to control() is a handle to an algorithm instance.

The second argument is an algorithm specific control command. See IALG_CmdId enumeration for details.

‖ Preconditions

The following conditions must be true prior to calling this function; otherwise, its operation is undefined.

- `control()` can only be called after a successful return from `algInit()` and `algActivate()`.
- If algorithm uses DMA resources, `control()` can only be called after a successful return from `DMAN3_init()`.
- `handle` must be a valid handle for the algorithm's instance object.
- params must not be NULL and must point to a valid `IALG_Params` structure.

‖ Postconditions

The following conditions are true immediately after returning from this function.

- If the control operation is successful, the return value from this operation is equal to `IALG_EOK;` otherwise it is equal to either `IALG_EFAIL` or an algorithm specific return value. If status or handle is NULL then TIOP returns `IALG_EFAIL`
- If the control command is not recognized or some parameters to act upon are not supported, the return value from this operation is not equal to `IALG_EOK`.
- The algorithm should not modify the contents of params. That is, the data pointed to by this parameter must be treated as read-only.

‖ Example

See test bench file, tiop_tb.c available in the \test\src sub-directory.

‖ See Also

algInit(), algActivate(), process()

5.7 Data Processing API

Data processing API is used for processing the input data.

‖ Name

`algActivate()` – initialize scratch memory buffers prior to processing.

‖ Synopsis

```
void algActivate(IALG_Handle handle);
```

‖ Arguments

IALG_Handle handle; /* algorithm instance handle */

‖ Return Value

Void

Description

`algActivate()` initializes any of the instance's scratch buffers using the persistent memory that is part of the algorithm's instance object.

The first (and only) argument to `algActivate()` is an algorithm instance handle. This handle is used by the algorithm to identify various buffers that must be initialized prior to calling any of the algorithm's processing methods.

For more details, see *TMS320 DSP Algorithm Standard API Reference.* (literature number SPRU360).

‖ See Also

`algDeactivate()`

‖ Name

`process()` – basic encoding/decoding call

‖ Synopsis

XDAS_Int32 (*process)(IVISION_Handle handle, IVISION_inBufs *inBufs, IVISION_outBufs *outBufs, IVISION_InArgs *inargs, IVISION_OutArgs *outargs);

‖ Arguments

IVISION_Handle handle; /* algorithm instance handle */

IVISION_inBufs *inBufs; /* algorithm input buffer descriptor */

IVISION_outBufs *outBufs; /* algorithm output buffer descriptor */

IVISION_InArgs *inargs /* algorithm runtime input arguments */

IVISION_OutArgs *outargs /* algorithm runtime output arguments */

‖ Return Value

IALG_EOK; /* status indicating success */

IALG_EFAIL; /* status indicating failure */

‖ Description

This function does the TI Object propagation. The first argument to `process()` is a handle to an algorithm instance.

The second and third arguments are pointers to the input and output buffer descriptor data structures respectively (see `IVISION_inBufs, IVISION_outBufs` data structure for details).

The fourth argument is a pointer to the `IVISION_InArgs` data structure that defines the run time input arguments for an algorithm instance object.

The last argument is a pointer to the `IVISION_OutArgs` data structure that defines the run time output arguments for an algorithm instance object.

> Note:
>
> If you are using extended data structures, the fourth and fifth arguments must be pointers to the extended `InArgs` and `OutArgs` data structures respectively. Also, ensure that the `size` field is set to the size of the extended data structure. Depending on the value set for the `size` field, the algorithm uses either basic or extended parameters.

‖ Preconditions

The following conditions must be true prior to calling this function; otherwise, its operation is undefined.

- `process()` can only be called after a successful return from `algInit()`.
- If algorithm uses DMA resources, `process()` can only be called after a successful return from `DMAN3_init()`.
- `handle` must be a valid handle for the algorithm's instance object.
- Buffer descriptor for input and output buffers must be valid.
- Input buffers must have valid input data.
- `inBufs->numBufs` indicates the total number of input
- Buffers supplied for input frame, and conditionally, the algorithms meta data buffer.
- `inArgs` must not be NULL and must point to a valid `IVISION_InArgs` structure.
- `outArgs` must not be NULL and must point to a valid `IVISION_OutArgs` structure.
- `inBufs` must not be NULL and must point to a valid `IVISION_inBufs` structure.
- `inBufs->bufDesc[0].bufs` must not be NULL, and must point to a valid buffer of data that is at least `inBufs->bufDesc[0].bufSize` bytes in length.
- `outBufs` must not be NULL and must point to a valid `IVISION_outBufs` structure.
- `outBufs->buf[0]` must not be NULL and must point to a valid buffer of data that is at least `outBufs->bufSizes[0]` bytes in length.
- The buffers in `inBuf` and `outBuf` are physically contiguous and owned by the calling application.

‖ Postconditions

The following conditions are true immediately after returning from this function.

- If the process operation is successful, the return value from this operation is equal to `IALG_EOK;` otherwise it is equal to either `IALG_EFAIL` or an algorithm specific return value.
- The algorithm must not modify the contents of `inArgs`.
- The algorithm must not modify the contents of `inBufs`, with the exception of `inBufs.bufDesc[].accessMask`. That is, the data and buffers pointed to by these parameters must be treated as read-only.
- The algorithm must appropriately set/clear the `bufDesc[].accessMask` field in `inBufs` to indicate the mode in which each of the buffers in `inBufs` were read. For example, if the algorithm only read from `inBufs.bufDesc[0].buf` using the algorithm processor, it could utilize `#SETACCESSMODE_READ` to update the appropriate `accessMask` fields. The application may utilize these returned values to manage cache.

- The buffers in `inBufs` are owned by the calling application.

‖ Example

See test application file, tiop_tb.c available in the \test\src sub-directory.

‖ See Also

algInit(), algDeactivate(), control()

> Note:
>
> The algorithm cannot be preempted by any other algorithm instance. That is, you cannot perform task switching while filtering of a particular frame is in progress. Pre-emption can happen only at frame boundaries and after `algDeactivate()` is called.

‖ Name

`algDeactivate()` – save all persistent data to non-scratch memory

‖ Synopsis

Void algDeactivate(IALG_Handle handle);

‖ Arguments

IALG_Handle handle; /* algorithm instance handle */

‖ Return Value

Void

‖ Description

`algDeactivate()` saves any persistent information to non-scratch buffers using the persistent memory that is part of the algorithm's instance object.

The first (and only) argument to `algDeactivate()` is an algorithm instance handle. This handle is used by the algorithm to identify various buffers that must be saved prior to next cycle of `algActivate()` and processing.

For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

‖ See Also

`algActivate()`

5.8 Termination API

Termination API is used to terminate the algorithm instance and free up the memory space that it uses.

‖ Name

`algFree()` – determine the addresses of all memory buffers used by the algorithm

‖ Synopsis

XDAS_Int32 algFree(IALG_Handle handle, IALG_MemRec memTab[]);

‖ Arguments

IALG_Handle handle; /* handle to the algorithm instance */

IALG_MemRec memTab[]; /* output array of memory records */

‖ Return Value

XDAS_Int32; /* Number of buffers used by the algorithm */

‖ Description

`algFree()` determines the addresses of all memory buffers used by the algorithm. The primary aim of doing so is to free up these memory regions after closing an instance of the algorithm.

`The first argument to algFree() is a handle to the algorithm instance.`

The second argument is a table of memory records that describe the base address, size, alignment, type, and memory space of all buffers previously allocated for the algorithm instance.

For more details, see *TMS320 DSP Algorithm Standard API Reference* (literature number SPRU360).

‖ See Also

`algAlloc()`

6 Frequently Asked Questions

This chapter provides answers to few frequently asked questions related to using this algorithm.

6.1 Code Build and Execution

Question	Answer

6.1.1 Algorithm Related

Question	Answer