These notes cover general changes and issues found in the MPF code base. Specifically, there are changes from Beta 2 that will be in effect in the next release.

The IVsBuildManagerAccessor Interface

1. The method IsBuildInProgress has been removed. This was incorrect in many cases.
2. The BeginOneOffBuid and EndOneOffBuild are now BeginDesignTimeBuild and EndDesignTimeBuild respectively.
3. Since this change was made after Beta 2, the changes are not in the Beta 2 code base. Code below indicates how the new methods would be used in a build after Beta 2.

THREADING RULES

1. You must NEVER invoke the MSBuild APIs on the BuildManager directly from within the IDE except as noted below.  You must use the BuildManagerAccessor in CommoIDE to mediate MSBuild, which is a shared resource.
2. If you invoke builds asynchronously, you need to ensure you don’t invoke additional MSBuild BuildManager APIs from within the callbacks you provide to MSBuild.
3. If your build process changes environment variables, you must set the NodeAffinity for your build to OutOfProc to avoid polluting the VS environment.  This also means that you cannot have a host object since that requires the in-proc node.  Use HostServices to do this.

 HOW TO BUILD WITH MSBUILD INSIDE VISUAL STUDIO

In Visual Studio 2010 with MSBuild 4.0, there are new interactions between the Solution Build Manager and MSBuild which impact project systems utilizing these services.  MSBuild 4.0 contains a new component called the Build Manager (not to be confused with the Solution Build Manager which is a VS component) which controls the submission of build requests.  This became necessary as Visual Studio 2010 now allows parallel builds to occur (notably native projects) and access to shared resources such as the CPU needed to be mediated.

For project systems which previously simply called Project.Build() to invoke a build several changes must be made.  The project system now must:

1. Ask for the SVsBuildManagerAccessor service using the IServiceProvider interface.  This should be done soon after the project system is loaded, well before any builds might occur.
2. Notify the system if you need the UI thread
3. Notify the system if you are doing a design-time build.
4. Register its loggers using the Build Manager Accessor.
5. Submit build requests directly to the MSBuild Build Manager, rather than invoking a method on the Project.

Here is an example of how all of this can be accomplished.  I will assume that the project system already has a reference to an IServiceProvider...

public bool Build(Project project, bool isDesignTimeBuild)

{

IVsBuildManagerAccessor accessor =

serviceProvider.GetService(typeof(SVsBuildManagerAccessor)) as

IVsBuildManagerAccessor;

bool releaseUIThread = false;

try

{

if(accessor != null)

{

// Need to claim the UI thread for build under the following conditions:

// 1. The build must use a resource that uses the UI thread, such as

// - you set HostServices and you have a host object which requires (even

// indirectly) the UI thread (VB and C# compilers do this for instance.)

// or,

// 2. The build requires the in-proc node AND waits on the UI thread for the

// build to complete, such as:

// - you use a ProjectInstance to build, or

// - you have specified a host object, whether or not it requires the UI

// thread, or

// - you set HostServices and you have specified a node affinity.

// - In addition to the above you also call submission.Execute(), or you call

// submission.ExecuteAsync() and then also submission.WaitHandle.Wait\*().

if (/\* either of the above conditions are met \*/)

{

int result = accessor.ClaimUIThreadForBuild();

if (result != S\_OK)

{

// Not allowed to claim the UI thread right now. Try again later.

return false;

}

releaseUIThread = true;

}

if (isDesignTimeBuild)

{

int result = accessor.BeginDesignTimeBuild();

if (result != S\_OK)

{

// Not allowed to begin a design-time build at this time. Try again later.

return false;

}

}

}

bool buildSucceeded = false;

/\* perform project-system specific build set up tasks

\* Create your BuildRequestData

\*/

IHostServices hostServices = /\* your host services, if any \*/;

// If you don't use a project instance (you build from a file for example) then

// you would use another appropriate constructor.

BuildRequestData requestData = new BuildRequestData(

project.CreateProjectInstance(), "myTarget", hostServices,

BuildRequestData.BuildRequestDataFlags.None);

// Pend your submission

BuildSubmission submission =

BuildManager.DefaultBuildManager.PendBuildRequest(requestData);

// Register your loggers

if (accessor != null)

{

foreach (ILogger logger in /\* your list of loggers \*/)

{

accessor.RegisterLogger(submission.SubmissionId, logger);

}

}

// call submission.Execute or submission.ExecuteAsync()

// Wait for build to complete as appropriate

submission.Execute();

BuildResult buildResult = submission.Execute();

// Set buildSucceeded as appropriate

return buildSucceeded;

}

finally

{

if (accessor != null)

{

// Unregister the loggers, if necessary.

accessor.UnregisterLoggers(submission.SubmissionId);

if (releaseUIThread)

{

accessor.ReleaseUIThreadForBuild();

}

if (isDesignTimeBuild)

{

accessor.EndDesignTimeBuild();

}

}

}

}

**DO NOT LOG SYNCHRONOUSLY TO THE UI**

Loggers in Visual Studio 2010 are necessarily not run on the UI thread as they were in Orcas. The UI thread is not always pumping messages during the build, so if your logger is trying to synchronously marshal a call to the output window, it will jam; and that would prevent the build from finishing. What VB/C#, C++, F# and TeamArch have done is to put errors into a queue of some sort instead, and service the queue when on the main thread. This gives you pretty bursty logging but it’s unavoidable given how VS currently works.