Contrôleur AMX de la série NX : Recommandations de programmation

Histoire

Révision	Date	Créateur	Description
a	08.07.2024	PW	Pemière édition

Produits

Contrôleure AMX NX
Pavés de contrôle AMX MCP
AMX DVX/DGX ave contrôleur NX

Nom du fichier

AROCOM_TN0019a_AMX_NX_Recommandations_de_programmation_FR

Table des matières

Documents référencés

Ce document TN0019 se réfère, entre autres, aux documents suivants publiés par AMX :

- NetLinx.LanguageReferenceGuide.pdf
  - Chapitre "Comprendre quand DEFINE_PROGRAM s'exécute".
- NX-Series.WebConsole-ProgrammingGuide.pdf
  - Chapitre "Différences dans l'exécution du programme DEFINE_PROGRAM".
- NX_Controller_Best_Practices_for_Longevity.pdf

Introduction

Lors de la programmation des contrôleurs AMX de la série NX, il faut faire attention à certains détails qui étaient moins importants pour les types précédents, les contrôleurs NI. Ce document ne doit pas être considéré comme une simple recommandation. Certaines des instructions qu'il contient doivent être impérativement respectées afin de garantir la fiabilité éprouvée des contrôleurs AMX.

Variables

Variables globales

Les variables globales qui sont définies dans la section DEFINE_VARIABLE sont par défaut NON_VOLATILE. Cela signifie qu'elles ne sont pas stockées dans la RAM rapide, mais dans la NVRAM.

On appelle "NVRAM" (Non-Volatile) la mémoire qui est conservée sans alimentation électrique. Cette dernière est toutefois un peu plus lente et nettement plus petite.
Dans la plupart des applications, les variables NON_VOLATILE ne sont pas nécessaires. Il est donc recommandé de toujours définir les variables dans la section DEFINE_VARIABLE comme VOLATILE. La définition NON_VOLATILE ne devrait vraiment être utilisée que si elle est inévitable.

Exemple:

```
DEFINE_VARIABLE
VOLATILE      INTEGER nCounter
VOLATILE      CHAR    nLabels[3][50]
NON_VOLATILE INTEGER nStartupMode
VOLATILE      FLOAT   fGainLevel
VOLATILE      INTEGER nMicMute[8]
VOLATILE      ...
VOLATILE      ...
VOLATILE      ...
```

Remarque:

Les deux variantes suivantes de définition de variables ont le même effet, car les variables globales sont NON_VOLATILE par défaut.

```
DEFINE_VARIABLE
NON_VOLATILE INTEGER nStartupMode
INTEGER nStartupMode
```

Constantes

Les constantes sont définies dans la section DEFINE_ CONSTANT et sont toujours stockées dans la RAM.
Il n'est donc pas nécessaire de les définir comme VOLATILE ou NON_VOLATILE.

Variables statique locales

Les variables statiques locales, appelées LOCAL_VAR dans Netlinx, sont utilisées dans

- Des événements
- Des fonctions
- d'autres scopes non globaux, comme par exemple des blocs de code entre accolades {...} à l'intérieur de DEFINE_PROGRAM ou DEFINE_START.

Par défaut, ces variables sont stockées dans la NVRAM (NVRAM = plus lente et plus petite). C'est pourquoi il est recommandé de définir également les variables LOCAL_VAR comme VOLATILE, si elles ne doivent pas obligatoirement être conservées.

Exemple:

```
DATA_EVENT[vdvPROJ]
  {
  ONLINE:
    {
    LOCAL_VAR VOLATILE INTEGER nForLoop        // RAM
    LOCAL_VAR INTEGER nCounter                 // NVRAM
    FOR(nForLoop=1; nForLoop<=12; nForLoop++)
      {
      ...
      }
    nCounter++
    }
  }
```

L'exemple montre que la variable nForLoop est volatile. Non-Volatile n'aurait pas de sens ici, car la variable nForLoop se voit attribuer la valeur 1 à chaque fois dans l'instruction FOR.

Notez que

Les variables locales, non volatiles, conservent leur valeur après un démarrage. Dans l'exemple, la variable nCounter compte chaque événement en ligne de vdvPROJ après le démarrage et n'est pas mise à 0. Il n'est pas possible d'attribuer une valeur d'initialisation aux variables locales:

```
LOCAL_VAR INTEGER nCounter = 5     // = Error
```

Il n'est donc pas possible d'attribuer une valeur définie à la variable nCounter après un démarrage ! Cela devrait toujours être pris en compte lors de la création de code Netlinx.

BUTTON_EVENT

Pour BUTTON_EVENT, la déclaration PUSH : et la déclaration RELEASE : doivent obligatoirement être présentes. La déclaration HOLD : est facultative.

Si une déclaration PUSH : ou une déclaration RELEASE : n'est pas nécessaire, elle doit tout de même être mentionnée dans le code avec des parenthèses vides.

L'exemple suivant montre un BUTTON_EVENT pour lequel le code n'est exécuté que dans l'énoncé PUSH : et l'énoncé RELEASE : n'est pas nécessaire :

```
BUTTON_EVENT[dvTP,0]
   {
   PUSH:
      {
      PULSE[dvRelay,1]
      }
   RELEASE:{}
}
```

CHANNEL_EVENT

Pour CHANNEL_EVENT, l'énoncé ON : et l'énoncé OFF : doivent obligatoirement être présents.

Si une déclaration ON : ou une déclaration OFF : n'est pas nécessaire, elle doit tout de même être mentionnée dans le code avec des parenthèses vides.

L'exemple suivant montre un CHANNEL_EVENT dans lequel le code n'est exécuté que dans l'énoncé ON : et l'énoncé OFF : n'est pas nécessaire :

```
CHANNEL_EVENT[dvIO,3]
   {
   ON:
      {
      PULSE[dvRelay,1]
      }
   OFF:{}
}
```

DEFINE_PROGRAM

VARIABLES

Il faut impérativement éviter de modifier les variables globales dans la section DEFINE_PROGRAM comme dans l'exemple suivant :

```
DEFINE_PROGRAM
nStatuscounter = nMode+1                    // Ne faites pas ça !!!
```

L'interrogation de variables globales comme dans l'exemple suivant n'est toutefois pas critique :

```
DEFINE_PROGRAM
[dvTP,201] = (nCurrentInput = 1)
[dvTP,202] = (nCurrentInput = 2)
[dvTP,203] = (nCurrentInput = 3)
[dvTP,204] = (nCurrentInput = 4)
```

Néanmoins, dans cet exemple également, il est recommandé d'exécuter le code dans un WAIT.

WAIT

En général, il est fortement recommandé que le code dans la section DEFINE_PROGRAM soit toujours exécuté dans un WAIT (par exemple un WAIT 1 = 100ms). L'exemple suivant n'est pas critique, bien que la variable nINC soit modifiée dans DEFINE_PROGRAM:

```
DEFINE_PROGRAM
WAIT 1
  {
  STACK_VAR INTEGER nINC
  FOR(nINC=1;nINC<=8;nINC ++)
    {
    [dvTP,200+INC] = (nCurrentInput = nINC)
    }
  }
```

Cet exemple n'est donc pas critique parce que

- la variable nINC n'est valable que dans le scope de WAIT 1 (la variable n'est pas globale).
- la variable nINC est définie comme STACK_VAR et est donc stockée dans la RAM.
- ce code n'est exécuté que toutes les 100 ms

TIMELINE

Au lieu d'un WAIT dans la section DEFINE_PROGRAM, il est également possible d'utiliser une TIMELINE, comme le montre l'exemple suivant :

```
DEFINE_CONSTANT
TL2_ID_FEEDBACK = 2



DEFINE_VARIABLE
LONG TL2_INTERVAL_FEEDBACK[] = {100}      // 100ms



DEFINE_START
TIMELINE_CREATE(TL2_ID_FEEDBACK, TL2_INTERVAL_FEEDBACK, 1,
                              TIMELINE_RELATIVE, TIMELINE_REPEAT)


DEFINE_EVENT
TIMELINE_EVENT[TL2_ID_FEEDBACK]           // Timeline-2: 100ms
   {
    insérer ici le code qui doit être exécuté toutes les 100 ms,
    par exemple, le feedback des boutons sur l'écran tactile

   IF(TIMELINE.REPETITION % 5 = 0)
      {
       insérer ici le code qui doit être exécuté toutes les 500ms
       }
   }
```

SEND_STRING 0

Une méthode populaire pour afficher des informations de débogage est d'utiliser SEND_STRING 0.

Il est important de savoir que les messages SEND_STRING 0 sont également enregistrés comme AUDIT LOG. Les AUDIT LOGs sont enregistrés sur la carte SD. Une carte SD peut traiter un grand nombre de cycles d'écriture, mais pas à l'infini. Cela signifie qu'une utilisation excessive de SEND_STRING 0 peut endommager la carte SD à long terme.

AMX_LOG()

L'exemple suivant montre la commande alternative AMX_LOG() au lieu de la commande SEND_STRING 0,"'Hello World'" :

```
AMX_LOG(AMX_DEBUG,"'Hello World'")
```

Pour voir le message 'Hello World' dans Diagnostics, il faut d'abord régler le LOG LEVEL sur DEBUG. Pour ce faire, il existe deux possibilités :

1. Dans le logiciel Netlinx, par exemple, dans la section DEFINE_START avec la commande `SET_LOG_LEVEL(AMX_DEBUG)`

2. Dans Telnet ou SSH avec la commande
   `netlinx log level` ... puis sélectionner le niveau 4 pour le débogage

Périphérique virtuel

Une autre possibilité d'éditer des informations de débogage est d'utiliser n'importe quel périphérique virtuel, dont aucun DATA_EVENT ne doit toutefois être implémenté.

Arocom utilise pour cela par défaut le périphérique virtuel suivant :

```
DEFINE_DEVICE
vdvDebOutp = 33011:01:0      // Sortie de débogage
                                Informations sous forme de
                                SEND_STRING
```

Modules

Il ne faut pas oublier que toutes ces mesures s'appliquent également aux modules et aux fichiers include !
Si les modules ne sont disponibles que sous forme compilée, il est impératif de connaître également le code source correspondant et d'en vérifier l'exactitude.

AUDIT LOG

Outre la commande SEND_STRING 0 déjà mentionnée, le système d'exploitation Netlinx enregistre aussi directement différents événements dans les AUDIT LOGs. Selon les fonctionnalités du logiciel, cela peut se produire assez fréquemment. Par exemple, si un contrôleur tente souvent d'ouvrir une connexion IP vers un appareil qui n'est pas présent sur le réseau, chaque erreur IP qui en résulte est enregistrée dans les AUDIT LOGs.

À partir de la version 1.6.205 du firmware maître, il est possible de désactiver l'enregistrement de ces événements. C'est pourquoi nous recommandons d'utiliser cette version de firmware ou une version supérieure.

FTP Write

L'écriture de fichiers sur l'espace FTP à l'aide des commandes FILE_WRITE et FILE_WRITE_LINE devrait être minimisée. En aucun cas, des fichiers ne devraient être écrits régulièrement sur l'espace FTP dans une boucle rapide.

Cas particulier du MCP

Aucune NVRAM n'est intégrée dans les produits de la série MCP. Il faut toujours en tenir compte lors de la création de code Netlinx.
Cela signifie que dans le code Netlinx pour MCP, cela n'a aucun sens de déclarer des variables comme NON_VOLATILE, car elles se comporteront comme des variables VOLATILE.
Il faut également en tenir compte lors de la création de modules Netlinx. Si des variables NON_VOLATILE doivent impérativement être utilisées dans un module, ce dernier n'est pas compatible avec les MCP.

SWITCH...CASE

Dans l'énoncé SWITCH...CASE, il faut veiller à ce que l'expression SWITCH n'accède pas à un tableau dont l'index n'est pas valable. Un tel accès ne se produira certes pas si la programmation est correcte, mais il se produit toujours dans la pratique.

Avec les contrôleurs NX, cette erreur entraîne dans certains cas un redémarrage, ce qui n'était pas le cas avec les modèles précédents de la série NI. Il convient donc d'être prudent, en particulier lors de l'adaptation d'une solution existante et ancienne à un contrôleur NX.

L'exemple suivant n'est pas critique, car nMode n'est pas un tableau :

```
STACK_VAR INTEGER nMode
...
SWITCH(nMode)
  {
  CASE 0:  { ...}
  CASE 1:  { ...}
  DEFAULT: { ...}
  }
```

Dans l'exemple suivant, la variable nIndex ne doit en aucun cas atteindre la valeur 0 ou une valeur >6, car le tableau nSetup[] ne fait que 6. Si ce code est exécuté dans une fonction, cela provoque un crash du contrôleur :

```
DEFINE_FUNCTION Test()
  {
  STACK_VAR INTEGER nIndex
  STACK_VAR INTEGER nSetup[6]
  ...
  SWITCH(nSetup[nIndex])
    {
    CASE 0:  { ...}
    CASE 1:  { ...}
    DEFAULT: { ...}
    }
  }
```

Pour éviter cela, l'expression dans le SWITCH peut être enregistrée au préalable dans une variable. Cette variable sera ensuite interrogée dans le SWITCH:

```
DEFINE_FUNCTION Test()
  {
  STACK_VAR INTEGER nIndex
  STACK_VAR INTEGER nSetup[6]
  STACK_VAR INTEGER nTemp
  ...
  nTemp = nSetup[nIndex]
  SWITCH(nTemp)
    {
    CASE 0:  { ...}
    CASE 1:  { ...}
    DEFAULT: { ...}
    }
  }
```

DIAGNOSTICS

Il est recommandé de prêter suffisamment attention à l'édition DIAGNOSTICS de NetLinxStudio et de la consulter régulièrement.