# Bibliothèques d'exécution personnalisées pour cartes nues

Cette annexe décrit comment personnaliser, construire et installer des bibliothèques d'exécution pour une utilisation sur des cibles de type carte nue.

GNAT Pro inclut une ou plusieurs bibliothèques d'exécution préinstallées pour une carte spécifique de l'ISA et de l'ABI ciblés par le compilateur et les outils auxiliaires. Dans certains cas, plusieurs cartes avec plusieurs bibliothèques d'exécution peuvent être supportées.

Les fichiers nécessaires à la construction de ces bibliothèques d'exécution sont inclus dans l'installation et peuvent être modifiés par les développeurs. Les fichiers peuvent être utilisés pour créer un run-time personnalisé, soit comme une modification relativement mineure d'une bibliothèque existante, soit comme une nouvelle bibliothèque d'exécution distincte pour une nouvelle carte.

Remarque : lorsque vous apportez des modifications permanentes à une bibliothèque d'exécution, assurez-vous de pouvoir répéter le processus, car vous devrez peut-être le faire plusieurs fois. En particulier, vous devrez peut-être le refaire lorsque vous passerez à une version plus récente du compilateur, car le compilateur et la bibliothèque d'exécution sont étroitement liés. Par conséquent, le temps d'exécution peut avoir changé. Si tel est le cas, vous devrez réappliquer vos modifications à cette nouvelle instance de la bibliothèque d'exécution. Par conséquent, nous vous suggérons de gérer les différences, ainsi que les fichiers qui en résultent, sous le contrôle de la configuration. Par exemple, vous pouvez générer et conserver des fichiers "patch" qui peuvent être réappliqués aux fichiers de la bibliothèque d'exécution.

La bibliothèque d'exécution est constituée de fichiers qui dépendent de la carte cible, ainsi que de fichiers qui sont indépendants de la cible. Ces derniers implémentent des fonctionnalités définies par le langage, tandis que les premiers sont responsables de la configuration et de l'interaction avec le matériel.

Les fichiers spécifiques à la cible sont connus sous le nom de BSP (Board Support Package). En tant que tels, ils peuvent nécessiter des modifications lors de l'adaptation d'une bibliothèque d'exécution pour une cible existante ou lors du portage de la bibliothèque d'exécution sur une nouvelle cible. En revanche, les fichiers implémentant la fonctionnalité définie par le langage ne seront presque jamais modifiés.

Les fichiers BSP sont situés dans les répertoires sources de chaque bibliothèque d'exécution. Les modifications apportées aux fichiers d'une bibliothèque d'exécution n'affectent pas les autres bibliothèques.

Pour adapter ou porter une bibliothèque d'exécution :

1. Copiez l'arborescence d'une bibliothèque d'exécution existante
2. Modifiez ou augmentez les fichiers de la nouvelle bibliothèque d'exécution.
3. construire la bibliothèque d'exécution.

Les sections ci-dessous fournissent des détails et un exemple de réalisation de ces étapes.

## Raison d'être d'une nouvelle bibliothèque d'exécution

Les sous-sections suivantes abordent les différentes considérations qui peuvent conduire à la création d'une nouvelle bibliothèque d'exécution.

### Considérations relatives au matériel

Les bibliothèques d'exécution testées et livrées avec GNAT sont destinées à des cartes et des configurations spécifiques. Étant donné que les microcontrôleurs ont une gamme d'options de configuration sur la puce, par exemple des configurations de mémoire et d'horloge, une bibliothèque d'exécution fournie supportant votre microcontrôleur peut ne pas être configurée de manière optimale pour votre carte spécifique.

Par exemple, les run-times STM32F4 fournis avec GNAT sont configurés pour supporter le STM32F4 Discovery kit et son microcontrôleur STM32F407. Même si vous utilisez le STM32F407 dans votre propre projet, vous pouvez avoir besoin de personnaliser le run-time STM32F4 pour supporter votre configuration particulière. Par exemple, les run-times de la STM32F4 sont configurés pour utiliser l'horloge externe de 8MHz fournie par le kit de découverte STM32F4 alors que votre installation peut utiliser une source d'horloge différente, comme l'oscillateur RC interne.

Cependant, tous les run-times fournis ne sont pas nécessairement liés à un microcontrôleur particulier. En poursuivant l'exemple du STM32F4, les run-times STM32F4 supportent les kits STM32F4 Discovery et STM32F429 Discovery, qui partagent des configurations matérielles similaires. Mais sans adapter le run-time du STM32F4, vous ne pouvez pas profiter des fréquences d'horloge plus élevées et de la plus grande mémoire disponibles sur le microcontrôleur STM32F429.

On pourrait modifier la bibliothèque d'exécution pour le F4 afin de supporter le F429, mais alors elle ne fonctionnerait plus pour une cible F4 réelle. Dans ce cas, il est logique de créer une nouvelle bibliothèque d'exécution et de l'adapter spécifiquement au matériel distinctif, avec un nom indiquant la nouvelle cible.

D'autres différences matérielles justifient également un nouveau run-time. Par exemple, le nombre et la priorité des interruptions peuvent être différents, ou le code de démarrage peut être nécessaire pour installer des périphériques supplémentaires ou les configurer d'une manière différente. En général, les modifications qui ne fonctionneraient pas sur la cible d'un run-time fourni justifient une nouvelle bibliothèque de run-time.

### Considérations relatives aux logiciels

La dépendance des applications à l'égard des fonctionnalités définies par le langage est également une raison de créer une nouvelle bibliothèque d'exécution. Cette dépendance est un problème potentiel car tous les runtimes bareboard fournissent des sous-ensembles des fonctionnalités définies par le langage. Les différents moteurs d'exécution fournissent différents sous-ensembles. Par exemple, certains moteurs d'exécution n'incluent pas les paquets mathématiques standard d'Ada. Vous pouvez décider d'augmenter le sous-ensemble d'un run-time fourni en conséquence, auquel cas une nouvelle bibliothèque d'exécution est sans doute appropriée.

Plus précisément, vous pouvez ajouter des fonctionnalités du langage qui sont mises en œuvre par des unités de programme autonomes ne nécessitant pas d'intégration avec le reste de la bibliothèque d'exécution. Ces unités ne sont pas implémentées directement par le compilateur et ne font partie de la bibliothèque d'exécution que pour les mettre à la disposition du constructeur. Les paquets mathématiques définis par le langage en sont de bons exemples, tout comme certains des divers paquets conteneurs.

Vous ne pouvez pas ajouter de fonctionnalités définies par le langage qui ne sont pas prises en charge par les runtimes de Ravenscar-Full.

Une autre raison pour laquelle une nouvelle bibliothèque d'exécution est nécessaire concerne l'allocation dynamique de la mémoire. Les différents moteurs d'exécution l'implémentent différemment, et vous pouvez vouloir changer la façon dont cela est fait. Une nouvelle bibliothèque d'exécution est justifiée dans ce cas également, même si rien d'autre n'est modifié.

### La structure de la bibliothèque d'exécution

Les implémentations d'exécution sont divisées en deux parties principales. La partie "GNAT" implémente la partie séquentielle du langage qui n'est pas implémentée directement par le compilateur. La partie "GNARL" implémente la partie "tasking" du langage. Comme les runtimes ZFP n'incluent pas les tâches, il n'y a pas de partie GNARL dans ces bibliothèques d'exécution.

L'arborescence des répertoires des bibliothèques d'exécution reflète ces partitions logiques. Pour les moteurs d'exécution basés sur Ravenscar qui incluent l'un ou les deux profils de tâches (Ravenscar et/ou Jorvik), il existe à la fois un sous-répertoire gnat et un sous-répertoire gnarl, chacun contenant les fichiers sources correspondants. Pour les run-times ZFP, qui ne contiennent pas de tasking, il n'y a que le sous-répertoire gnat. Ces sous-répertoires sont situés à la racine de l'arborescence des répertoires d'exécution.

De plus, les scripts d'édition de liens fournis sont situés dans le sous-répertoire ld, également à la racine de l'arbre d'exécution.

Les fichiers qui constituent le BSP sont situés dans ces répertoires.

Pour chaque run-time, il existe d'autres sous-répertoires destinés à contenir les fichiers que vous avez modifiés, le cas échéant. (Initialement, ces sous-répertoires sont vides.) Les noms des répertoires ont un suffixe "\_user" pour refléter cette intention. Il y aura toujours des sous-répertoires gnat\_user et ld\_user. Pour les run-times Ravenscar, il y aura également un sous-répertoire gnarl\_user.

Tous les fichiers trouvés dans ces répertoires "utilisateur" sont automatiquement prioritaires sur ceux des autres répertoires lorsque la bibliothèque d'exécution est construite. Par conséquent, nous vous recommandons de copier les fichiers que vous avez l'intention de modifier dans ces sous-répertoires, plutôt que de les modifier sur place. Cette dernière approche fonctionnera, mais le fait de les séparer facilitera le suivi des modifications.

### Construction d'une bibliothèque d'exécution

En définitive, une bibliothèque d'exécution modifiée doit être construite et installée avant que la chaîne d'outils (compilateur, etc.) puisse l'utiliser pour créer une image exécutable.

Les options prédéfinies du constructeur définissant divers paramètres de commutation sont déclarées comme variables de scénario dans les fichiers de projet GNAT (fichiers "gpr") utilisés pour construire la bibliothèque d'exécution. Les options du chargeur sont contrôlées par une variable de scénario définie dans un fichier XML. Ces options sont discutées dans les sections dédiées ci-dessous. Les utilisateurs peuvent modifier ces définitions si nécessaire.

**Important : vous devez utiliser le compilateur croisé pour la cible spécifiée par le fichier de projet d'exécution. Si plusieurs compilateurs GNAT sont installés, assurez-vous que votre chemin d'accès à l'environnement est défini de manière à ce que le compilateur croisé de votre cible ait la priorité sur les autres compilateurs. Si la construction d'une bibliothèque d'exécution échoue, il s'agit probablement du problème.**

#### Construction

Les bibliothèques d'exécution sont construites à l'aide de l'outil gprbuild, avec un fichier de projet GNAT primaire spécifié comme paramètre principal. Ces fichiers de projet de construction primaires sont nommés comme suit :

* Pour les moteurs d'exécution Ravenscar, le nom est ravenscar\_build.gpr pour les moteurs d'exécution "Full" et "SFP".
* Pour les run-times ZFP, le nom est plutôt runtime\_build.gpr.

Ces fichiers de projet font référence à un ou plusieurs autres fichiers de projet, mais les utilisateurs peuvent ignorer les autres, sauf si leur contenu nécessite une modification.

Tous les fichiers de projet sont situés à la racine des sous-répertoires individuels contenant les sources d'exécution et d'autres fichiers pour chaque bibliothèque d'exécution individuelle.

Par exemple, pour construire le run-time Ravenscar-SFP pour la cible stm32f4 (STM32F4 Discovery kit), nous nous rendons dans le sous-répertoire de cette bibliothèque d'exécution spécifique, puis nous lançons la commande suivante :

$ gprbuild -P ravenscar\_build.gpr

La bibliothèque d'exécution construite est située dans l'arborescence du sous-répertoire contenant le fichier de projet GNAT.

Vous pouvez appliquer des commutateurs de ligne de commande gprbuild supplémentaires si nécessaire. Par exemple, pour que gprbuild utilise tous les processeurs de votre ordinateur pour construire le run-time, vous devez également spécifier l'option -j0 :

$ gprbuild -j0 -P runtime\_build.gpr

Reportez-vous à la documentation de gprbuild pour connaître tous les commutateurs disponibles.

Comme mentionné, diverses options de construction de projet sont définies dans les fichiers de projet. Ces options sont spécifiées via la variable de scénario BUILD du projet et sont utiles lors du débogage ou du profilage des run-times :

* Production : (par défaut) aucune information de débogage, optimisé, pas d'assertions.
* Débogage : informations de débogage, pas d'optimisation, pas d'assertions.
* Assert : les assertions dans le code de la bibliothèque d'exécution sont activées.
* Gnatcov : définit les drapeaux appropriés pour l'utilisation de l'outil gnatcov.

Pour spécifier la variable de scénario, incluez -XBUILD=<valeur> sur la ligne de commande lorsque vous invoquez gprbuild. Par exemple, pour construire le run-time avec des informations de débogage, utilisez :

$ gprbuild -j0 -P ravenscar\_build.gpr -XBUILD=Debug

Sachez que l'activation des assertions au sein du moteur d'exécution augmente de manière significative les coûts d'exécution et de mémoire. Nous vous recommandons de n'activer les assertions d'exécution que sur demande du support AdaCore.

Si vous souhaitez supprimer les artefacts de construction, utilisez l'outil gprclean, en spécifiant à nouveau le fichier principal du projet GNAT. De plus, appliquez le commutateur "-r" pour nettoyer "récursivement" car le projet principal fait référence à d'autres projets. Par exemple :

$ gprclean -r -P ravenscar\_build.gpr

#### Choix de l'emplacement

La construction d'une bibliothèque d'exécution l'installe également de manière efficace car tout ce que vous devez faire pour l'utiliser est de transmettre son emplacement au constructeur et aux autres outils. Aucun emplacement particulier n'est requis. Le constructeur sera capable de l'utiliser via les approches standard documentées dans la section Spécification de la bibliothèque d'exécution, y compris le commutateur :switch:--RTS sur la ligne de commande et l'attribut Runtime dans les fichiers de projet GNAT.

Cependant, comme l'indique cette section, certains emplacements sont plus pratiques que d'autres. En particulier, si une bibliothèque de runtime est placée au même endroit que les runtimes fournis avec le compilateur GNAT, le runtime peut être spécifié aux outils GNAT par son seul nom (par exemple, "ravenscar-full-stm32f4"). S'il est installé en dehors de ce dossier, les outils exigent le chemin complet du moteur d'exécution.

Les moteurs d'exécution fournis avec les compilateurs bareboard GNAT sont situés dans un sous-répertoire sous la racine de l'installation GNAT, structuré comme suit :

**<gnat-installation-root> :**

**\--> <target>**

**|--> lib**

**|--> gnat**

**|--> <run-time>**

**|--> ...**

**\--> <run-time>**

Dans l'exemple ci-dessus, "<target>" est le nom de l'ISA supporté par le compilateur, et est la même chaîne utilisée pour spécifier l'attribut Target dans le fichier de projet GNAT. Pour les cibles ARM EABI, par exemple, la cible est "arm-eabi", l'arbre est donc le suivant :

**<gnat-installation-root> :**

**\--> arm-eabi**

**|--> lib**

**|--> gnat**

**|--> ravenscar-full-stm32f4**

**|--> ravenscar-full-tms570**

**|--> ...**

**|--> ravenscar-sfp-stm32f4**

**|--> ravenscar-sfp-tms570**

**|--> ...**

**|--> zfp-stm32f4**

**|--> zfp-tms570**

**\--> ...**

Le compilateur PowerPC ELF aurait "powerpc-elf" pour le nom du dossier <target>, avec des noms de bibliothèque d'exécution tels que "ravenscar-full-mpc8641" et "zfp-mpc8641" pour indiquer l'exécution et la carte supportées.

Si vous reconstruisez un run-time pour appliquer simplement des commutateurs différents, invoquez le constructeur comme décrit ci-dessus et spécifiez le fichier de projet GNAT existant ainsi que les variables et valeurs de scénario prévues. Si vous souhaitez un nouvel ensemble de commutateurs - en fait un nouveau scénario - nous vous suggérons d'augmenter les fichiers de projet existants pour définir ce scénario et ses commutateurs, puis de construire. Ce faisant, vous favoriserez la répétabilité de la construction dans le futur.

### Exemple : Création d'un run-time STM32F429

Pour illustrer le processus de modification d'un run-time pour une carte spécifique, cette section illustre l'adaptation de la bibliothèque de run-time pour le kit de découverte STM32F4 en un nouveau run-time supportant le kit de découverte STM32F429.

Le STM32F429 Discovery kit est similaire au STM32F4 Discovery kit mais contient plus de mémoire et supporte une horloge plus rapide. Par conséquent, le nouveau run-time sera créé à partir de la bibliothèque existante du run-time STM32F4. Nous allons également baser notre run-time sur celui de Ravenscar-SFP, la nouvelle bibliothèque run-time sera donc appelée ravenscar-sfp-stm32f429 pour refléter la cible différente.

#### Création initiale de la bibliothèque

Toute bibliothèque d'exécution bareboard fournie avec GNAT peut être utilisée comme base pour une nouvelle bibliothèque d'exécution personnalisée. Vous devez évidemment choisir une bibliothèque d'exécution ayant la même ISA et ABI (par exemple, powerpc-elf), mais plus la configuration de la bibliothèque d'exécution existante est proche de votre plate-forme, mieux c'est.

Nous vous suggérons de créer votre nouvelle bibliothèque d'exécution comme une copie d'une bibliothèque existante s'il en existe une suffisamment proche de votre plate-forme. Pour ce faire, copiez l'intégralité de l'arborescence des sous-répertoires d'exécution et apportez les modifications dans cette copie. Le fait de commencer à partir d'une copie vous permet d'utiliser ou de faire référence au run-time original sans avoir à réinstaller GNAT. Nous suggérons également que la copie soit située dans le système de fichiers avec les moteurs d'exécution fournis avec le compilateur, pour la commodité de référence décrite ci-dessus.

Contactez AdaCore s'il n'y a pas de run-time GNAT livré qui soit suffisamment proche pour être un bon point de départ.

En utilisant un shell Unix par exemple, nous allons d'abord dans le sous-répertoire GNAT contenant les run-times fournis :

$ cd <répertoire d'installation de GNAT>/arm-eabi/gnat/lib

Veillez à utiliser le chemin indiqué, sous le répertoire racine de votre installation GNAT. Il y a d'autres dossiers avec les mêmes noms, y compris certains qui ont les noms des run-times, donc une certaine attention est requise.

Ensuite, copiez toute l'arborescence des répertoires représentant la bibliothèque d'exécution existante :

$ cp -r ravenscar-sfp-stm32f4 ravenscar-sfp-stm32f429

Puis allez dans cette nouvelle arborescence de répertoires :

$ cd ravenscar-sfp-stm32f429

Nous allons d'abord nettoyer la bibliothèque, puis la construire, comme décrit dans la section Construire une bibliothèque d'exécution. Comme il s'agit d'un run-time Ravenscar, le nom du fichier principal du projet est ravenscar\_build.gpr comme indiqué ci-dessous :

$ gprclean -r -P ravenscar\_build.gpr

Notez le commutateur -r pour nettoyer récursivement.

Et maintenant la construction, juste pour vérifier que nous commençons avec quelque chose qui se construit avec succès :

$ gprbuild -P ravenscar\_build.gpr

Bien que vous puissiez commencer à utiliser ce run-time immédiatement, il est fonctionnellement équivalent au run-time original du STM32F4 à ce stade.

#### Adaptation de la bibliothèque

Après avoir vérifié que nous commençons avec un run-time qui peut être construit avec succès, nous pouvons maintenant commencer à faire des changements orientés matériel aux fichiers BSP situés dans notre copie. Nous recommandons de modifier les copies des fichiers BSP afin de copier d'une partie de notre run-time à une autre, plutôt que d'une autre bibliothèque de run-time.

##### Modification de la fréquence d'horloge

Le paquetage BSP System.BB.Parameters (situé dans gnat/s-bbpara.ads) déclare de nombreux paramètres utilisés par la bibliothèque d'exécution pour configurer le microcontrôleur et décrire les caractéristiques du microcontrôleur disponibles pour l'exécution. Cela inclut la source et la fréquence de l'horloge.

Lorsqu'un run-time prend en charge plusieurs cartes d'une même famille de processeurs, nombre de ces paramètres proviennent de paramètres spécifiques à la carte déclarés dans le paquetage System.BB.Board\_Parameters (dans le fichier gnat/s-bbbopa.ads).

Les run-times du STM32F4 illustrent cette structure. La fréquence d'horloge du processeur est définie par le numéro nommé Main\_Clock\_Frequency dans le paquet System.BB.Board\_Parameters :

**package System.BB.Parameters is**

**Clock\_Frequency : constant := Board\_Parameters.Main\_Clock\_Frequency;**

Le paquetage System.BB.Board\_Parameters pour le microcontrôleur STM32F4 est situé dans le run-time dans le fichier gnat/s-bbbopa.ads. Comme ce fichier a été copié depuis le run-time STM32F4, la valeur actuelle de Main\_Clock\_Frequency est celle de la fréquence maximale du STM32F4 Discovery kit :

**package System.BB.Board\_Parameters is**

**Main\_Clock\_Frequency : constant := 168\_000\_000;**

Cependant, nous aimerions que notre run-time STM32F429 exécute les applications à la vitesse maximale des cartes STM32F429 Discovery : 180MHz. Par conséquent, nous copions le fichier spec gnat/s-bbbopa.ads dans le sous-répertoire gnat\_user et nous y apportons la modification :

**package System.BB.Board\_Parameters is**

**Main\_Clock\_Frequency : constant := 180\_000\_000;**

Une fois reconstruit, le nouveau run-time exécutera les applications sur les cartes STM32F429 Discovery à la fréquence d'horloge supérieure.

##### Adaptation de l'agencement de la mémoire

Pour changer la configuration de la mémoire physique d'une carte, il faut modifier les scripts du linker situés dans le répertoire ld. Les noms des scripts reflètent leur objectif. Par exemple, les cibles PowerPC comprennent les éléments suivants :

* ram.ld
* qemu-ROM.ld

Lorsque des familles cibles sont prises en charge, comme la famille STM32F4, des scripts communs sont définis, avec des scripts supplémentaires les adaptant à des cartes spécifiques. Par exemple, les scripts de la cible STM32 comprennent les éléments suivants :

* common-RAM.ld
* common-ROM.ld

Ces scripts communs sont adaptés par des scripts de linker supplémentaires qui définissent les tailles et les emplacements réels des régions de mémoire définies dans les scripts communs. Par exemple, la cible STM32F4 contient ld/memory-map.ld qui décrit les tailles de mémoire et les emplacements de la RAM et de la Flash sur le STM32F407.

Pour notre nouvelle cible STM32F429 Discovery, les scripts de linker communs fonctionneront car leur disposition de la mémoire est adaptée au STM32F429.

Cependant, le STM32F429 a plus de Flash et de RAM que le STM32F407. Nous devrons donc mettre à jour le script de liaison spécifique à la carte pour profiter de la plus grande mémoire. Pour ce faire, nous copions le script de liaison ld/memory-map.ld dans ld\_user et le modifions pour refléter la quantité de mémoire disponible sur le STM32F429. La modification est celle de l'original :

**MEMORY{**

**flash (rx) : ORIGIN = 0x08000000, LENGTH = 1024K**

**sram (rwx) : ORIGIN = 0x20000000, LENGTH = 128K**

**...}**

vers :

**MEMORY{**

**flash (rx) : ORIGIN = 0x08000000, LENGTH = 2048K**

**sram (rwx) : ORIGIN = 0x20000000, LENGTH = 192K**

**...}**

Avec ce changement en place, nos applications seront en mesure de profiter de l'augmentation de la mémoire disponible sur le STM32F429. Notez que nous n'avons pas besoin de modifier l'origine de chaque région mémoire car elles sont communes entre le STM32F407 et le STM32F429.

##### Sélection des Linker Scripts

Plusieurs linker scripts peuvent être inclus dans un run-time pour prendre en charge différents scénarios de chargement et de disposition de la mémoire. Par exemple, les scripts pour l'ARM Cortex-M prennent en charge le chargement d'un programme en Flash (common-ROM.ld) ou en RAM (common-RAM.ld).

La variable de scénario LOADER sélectionne le script de linker utilisé lors de la construction d'un projet d'application avec gprbuild. La variable est définie dans un fichier XML nommé run-time.xml : un fichier qui fournit à gprbuild des paramètres de projet spécifiques à la cible et qui se trouve à la racine de chaque arborescence de répertoires d'exécution.

Comme on peut le voir dans la version abrégée de ce fichier XML, la variable de scénario LOADER contrôle le choix du linker script :

**type Loaders is ("ROM", "RAM", "USER");**

**Loader : Loaders := external("LOADER", "ROM");**

**...**

**package Linker is**

**for Required\_Switches use Linker'Required\_Switches &**

**("-Wl,-L${RUNTIME\_DIR(Ada)}/adalib",**

**"-nostartfiles", "-lc", "-lgnat",**

**"-L${RUNTIME\_DIR(ada)}/ld\_user",**

**"-L${RUNTIME\_DIR(ada)}/ld") &**

**Compiler.Common\_Required\_Switches;**

**case Loader is**

**when "ROM" =>**

**for Required\_Switches use Linker'Required\_Switches &**

**("-T", "common-ROM.ld");**

**when "RAM" =>**

**for Required\_Switches use Linker'Required\_Switches &**

**("-T", "common-RAM.ld");**

**when "USER" =>**

**end case;**

**end Linker;**

Vous pouvez spécifier la valeur de cette variable de scénario sur la ligne de commande lorsque vous invoquez gprbuild, contrôlant ainsi quel script est appliqué. La valeur par défaut (ROM) est utilisée si aucun argument explicite n'est spécifié, tandis que la valeur USER permet de spécifier un script de linker spécifique au projet dans le fichier gpr du projet.

Par exemple, étant donné un fichier de projet app.gpr décrivant l'application, nous pouvons invoquer gprbuild et fournir une ou plusieurs valeurs de variables de scénario :

$ gprbuild -P app.gpr -XLOADER=RAM

Ou prendre la valeur par défaut :

$ gprbuild -P app.gpr

##### Augmentation de la taille du tas

Notre STM32F429 cible a plus de mémoire que le STM32F4 mais nous n'avons pas besoin de changer le run-time pour le rendre disponible pour l'allocation dynamique de mémoire. Le "heap" recevra automatiquement la mémoire supplémentaire car il s'agit d'un run-time Ravenscar-SFP.

Reportez-vous à la section Allocation dynamique de la mémoire pour plus de détails sur la manière dont les moteurs d'exécution mettent en œuvre l'allocation et la désallocation dynamiques de la mémoire.

##### Configuration des interruptions

Lorsque vous adaptez un run-time à un nouveau processeur, la configuration des interruptions doit être revue car le type et le nombre d'interruptions peuvent avoir changé. Ceci est important même lors du portage d'un run-time au sein d'une même famille de microcontrôleurs car le nombre d'interruptions varie en fonction des périphériques offerts par un membre individuel de la famille. Les fichiers à examiner sont situés dans le dossier gnarl du répertoire racine du run-time.

Au minimum, les vecteurs de piège d'interruption et les noms d'interruption doivent être examinés. En général, les vecteurs d'interruption sont définis dans une table de vecteurs d'interruption située dans un fichier d'assemblage : gnarl/handler.s. Lors du portage au sein d'une famille de microcontrôleurs, il suffit de s'assurer que le nombre correct de vecteurs d'interruption est défini. Lors du portage vers une autre famille de processeurs, vous devrez vous référer au manuel de référence de cette famille pour déterminer comment définir la table des vecteurs d'interruption.

Les noms d'interruption et leurs valeurs Interrupt\_ID correspondantes sont déclarés dans le paquet Ada.Interrupts.Names (dans le fichier gnarl/a-intnam.ads). Ce fichier devra être modifié si les noms des interruptions ou le nombre d'interruptions diffèrent sur la nouvelle cible.

Voir le paquet System.BB.MCU\_Parameters (dans le fichier gnat/s-bbmcpa.ads) pour le nombre d'interruptions définies. Ce nombre doit refléter le contenu du paquetage Ada.Interrupts.Names.

Le STM32F429 Discovery kit utilisé dans cet exemple a plus d'interruptions que le STM32F4 Discovery kit, en raison des dispositifs supplémentaires, donc notre nouveau run-time nécessite une modification.

Nous copions gnat/s-bbmcpa.ads dans le dossier gnat\_user et changeons la valeur de Number\_of\_Interrupts de 81 à 92 :

**Number\_Of\_Interrupts : constant := 92;**

Nous copions ensuite gnarl/a-intnam.ads dans le dossier gnarl\_user et ajoutons ce qui suit :

**-- Flash global interruptFLASH\_Interrupt : constant Interrupt\_ID := 4;**

Notez que la version F4 de ce fichier saute l'interruption numéro 4.

Nous ajoutons ensuite les interruptions supplémentaires :

**-- UART 7 global interrupt  
UART7\_Interrupt : constant Interrupt\_ID := 82;**

**-- UART 8 global interrupt  
UART8\_Interrupt : constant Interrupt\_ID := 83;**

**-- SPI 4 global interrupt  
SPI4\_Interrupt : constant Interrupt\_ID := 84;**

**-- SPI 5 global interrupt  
SPI5\_Interrupt : constant Interrupt\_ID := 85;**

**-- SPI 6 global interrupt  
SPI6\_Interrupt : constant Interrupt\_ID := 86;**

**-- SAI1 global interrupt  
SAI1\_Interrupt : constant Interrupt\_ID := 87;**

**-- LTDC global interrupt  
LTDC\_Interrupt : constant Interrupt\_ID := 88;**

**-- LTDC global error interrupt  
LTDC\_ER\_Interrupt : constant Interrupt\_ID := 89;**

**-- DMA2D global interrupt  
DMA2D\_Interrupt : constant Interrupt\_ID := 90;**

De même, dans handler.S, nous ajoutons les entrées supplémentaires de la table de vecteurs de piège :

**.word \_\_gnat\_irq\_trap /\* 98 UART7\_IRQ \*/**

**.word \_\_gnat\_irq\_trap /\* 99 UART8\_IRQ \*/**

**.word \_\_gnat\_irq\_trap /\* 100 SPI4\_IRQ \*/**

**.word \_\_gnat\_irq\_trap /\* 101 SPI5\_IRQ \*/**

**.word \_\_gnat\_irq\_trap /\* 102 SPI6\_IRQ \*/**

**.word \_\_gnat\_irq\_trap /\* 103 SAI1 \*/**

**.word \_\_gnat\_irq\_trap /\* 104 LTDC \*/**

**.word \_\_gnat\_irq\_trap /\* 105 LTDC\_ER \*/**

**.word \_\_gnat\_irq\_trap /\* 106 DMA2D \*/**

Nous reconstruisons ensuite la bibliothèque pour mettre en place ces mises à jour.

##### Priorités d'interruption

Le nouveau matériel cible peut prendre en charge un nombre de priorités d'interruption matérielle différent de celui du processeur dans le run-time d'origine. Dans ce cas, le sous-type Interrupt\_Priority défini dans le paquetage System doit être modifié. (Prenez note des directives relatives à la modification des plages de priorité et de leurs exigences définies par le langage, telles que documentées dans cette section).

Le fichier du paquetage System se trouve dans gnat/system.ads. Son contenu dépend fortement du profil d'exécution (chaque profil d'exécution nécessite sa propre version).

Le paquetage System contient des déclarations de sous-types et de constantes prioritaires définies conformément au manuel de référence Ada. Le run-time STM32F4 Ravenscar-SFP, par exemple, contient les définitions suivantes :

**Max\_Interrupt\_Priority : constant Positive := 255;  
Max\_Priority : constant Positive := 240;**

**subtype Any\_Priority is Integer range 0 .. 255;  
subtype Priority is Any\_Priority range 0 .. 240;  
subtype Interrupt\_Priority is Any\_Priority range 241 .. 255;**

**Default\_Priority : constant Priority := 120;**

**Remarque : vous ne devez rien modifier d'autre dans le paquetage.**

Le manuel de référence Ada définit le sous-type Any\_Priority pour représenter toutes les valeurs de priorité, les chiffres les plus élevés indiquant des priorités plus importantes. Ce sous-type est divisé en deux sous-groupes : Priority et Interrupt\_Priority. Interrupt\_Priority correspond aux priorités utilisées par les gestionnaires d'interruptions, tandis que les tâches, y compris la tâche d'environnement, utilisent généralement les priorités du sous-type Priority. Cette affectation de priorités garantit que les interruptions peuvent interrompre l'exécution normale des tâches.

Bien qu'Ada permette d'attribuer aux tâches des priorités de la plage Interrupt\_Priority, cela n'est pas courant et peut être empêché sur certaines cibles, telles que ARM. (Les tâches ne sont plus destinées à être des gestionnaires d'interruptions, après la mise à jour initiale de la norme du langage Ada).

La plage 0 ... 255 est recommandée pour le sous-type Any\_Priority car, en pratique, 256 niveaux de priorité sont suffisants pour toute application et une plage plus petite ne réduit pas les besoins en mémoire de la bibliothèque d'exécution. Toutefois, vous pouvez utiliser les limites correspondant au matériel. Bien que la norme Ada exige que System.Priority comprenne au moins 30 valeurs, cela est soumis aux réalités matérielles, comme toutes les exigences de ce type.

La plage de System.Interrupt\_Priority doit correspondre à la plage des niveaux de priorité du matériel d'interruption de sorte qu'il y ait une correspondance univoque entre eux. Ainsi, si votre nouvelle cible supporte un nombre différent de niveaux de priorité d'interruption matérielle, mettez à jour Interrupt\_Priority pour refléter ce nombre et ajustez la plage de Priority en conséquence. Par exemple, si votre nouvelle cible ne prend en charge qu'une seule priorité d'interruption matérielle, mettez à jour System comme suit :

**Max\_Interrupt\_Priority : constant Positive := 255;  
Max\_Priority : constant Positive := 254;**

**subtype Any\_Priority is Integer range 0 .. 255;  
subtype Priority is Any\_Priority range 0 .. 254;  
subtype Interrupt\_Priority is Any\_Priority range 255 .. 255;**

**Default\_Priority : constant Priority := 127;**

Notre cible STM32F429 a les mêmes priorités d'interruption que le STM32F4, nous n'avons donc pas besoin de faire de changement dans cet exemple.

##### Code de démarrage

Le code de démarrage du STM32F4 est composé de plusieurs fichiers en langage assembleur situés dans le sous-répertoire gnat de la bibliothèque d'exécution.

Il y a deux fichiers en langage assembleur pour le code de démarrage, un pour l'exécution depuis la RAM ou la ROM, plus un fichier commun partagé par les deux (qui démarre le FPU, par exemple). Ces fichiers sont nommés start-ram.S, start-rom.S et start-common.S.

Le code de démarrage spécifique est sélectionné par les références des scripts du linker aux symboles uniques définis dans les fichiers d'assemblage, via les définitions des points d'entrée.

Pour les temps d'exécution de Ravenscar, la table vectorielle est initialisée par le code d'assemblage dans gnarl/handler.S. Le code Ada dans le paquetage System.BB.CPU\_Primitives (gnat/s-bbcppr.adb) installe des gestionnaires spécifiques à GNAT qui lèvent des exceptions pour les pièges. (Dans le run-time ZFP, le code de démarrage initialise la table vectorielle).

Notre cible STM32F429 a les mêmes exigences de démarrage que le STM32F4, nous n'avons donc pas besoin de faire de changements dans cet exemple.

##### Co-processeur à virgule flottante

Le paquet System.BB.Parameters (dans le fichier :file`gnat/s-bbpara.ads`) spécifie si une unité à virgule flottante (FPU) est présente, mais ceci est utilisé dans le code conditionnel de la routine de changement de contexte, et non pour décider d'activer ou non la FPU. Les cibles STM32 supportées ont toutes une FPU, donc l'unité est activée dans le code de démarrage commun (start-common.S). Par conséquent, nous n'avons pas besoin de modifier le code de démarrage pour notre cible F429.

#### Augmentation de la bibliothèque

Comme décrit dans la section Considérations logicielles, il se peut que vous souhaitiez créer un nouveau moteur d'exécution afin d'apporter des modifications au logiciel. Cette section est consacrée au cas spécifique de l'ajout de fonctionnalités logicielles en augmentant la bibliothèque d'exécution.

Nous vous recommandons de créer une nouvelle bibliothèque d'exécution dans ce but. Si vous l'avez déjà fait pour des modifications matérielles, vous continuerez à travailler avec cette nouvelle bibliothèque d'exécution dans cette section. Sinon, reportez-vous d'abord à la section Création initiale de la bibliothèque pour savoir comment créer une nouvelle bibliothèque d'exécution.

Notez que l'extension d'une bibliothèque de run-time n'a de sens que si votre nouveau run-time n'est pas une copie d'un run-time Ravenscar-Full. Sinon, le nouveau run-time contient déjà toutes les fonctionnalités possibles.

L'extension d'une bibliothèque d'exécution implique de copier des fichiers individuels à partir d'une bibliothèque d'exécution plus performante. Si vous augmentez un run-time ZFP, vous pouvez copier les fichiers de l'un ou l'autre des deux run-times Ravenscar. La copie à partir d'un runtime Ravenscar-Full peut être plus pratique car ces runtimes incluent toutes les entités définies en langage Ada que GNAT peut supporter dans un runtime bare-board. Si vous augmentez un run-time Ravenscar-SFP, seul un run-time Ravenscar-Full dispose de plus de fonctionnalités à copier.

N'importe quel code de run-time ne peut pas être copié dans une nouvelle bibliothèque. Vous devez respecter les limitations décrites dans la section Considérations logicielles.

Les fichiers sources candidats, et leurs dépendances, se trouvent dans le répertoire gnat. Ils doivent être copiés dans le répertoire gnat\_user de votre système d'exécution. Consultez la section Structure de la bibliothèque de l'environnement d'exécution si vous n'êtes pas familier avec ces répertoires.

Vous devez vous assurer que les dépendances, le cas échéant, sont également copiées dans le nouveau run-time. Comme vous le verrez plus loin, ces dépendances peuvent être nombreuses.

Comme il s'agit de code source d'exécution, il est possible que le code copié ait des exigences matérielles spécifiques. Cela est particulièrement vrai pour les unités de niveau inférieur dont dépend le code d'implémentation de la bibliothèque définie par le langage. Ces exigences doivent être satisfaites par le matériel réel utilisé par vos applications. Par exemple, les unités d'exécution implémentant la fonctionnalité de virgule flottante peuvent être implémentées dans l'attente d'un support matériel (par exemple, un coprocesseur). Vous devez vous assurer que le support en virgule flottante attendu est disponible. Sinon, choisissez un autre ensemble de fichiers d'implémentation à copier.

Une fois les sources supplémentaires copiées, il suffit de construire le run-time pour inclure la nouvelle fonctionnalité. Les unités définies par le langage peuvent alors être utilisées par l'application. Les dépendances de niveau inférieur ne doivent pas être référencées par l'application, généralement pour des raisons de portabilité.

**Remarque :** si la construction d'une application échoue avec un message d'erreur indiquant que le constructeur GNAT n'a pas connaissance d'une unité définie par le langage que vous pensez avoir ajoutée au run-time, soit vous n'avez pas copié les fichiers source corrects, soit vous n'avez pas reconstruit la bibliothèque du run-time après avoir copié les fichiers.

À titre d'exemple d'augmentation du temps d'exécution, disons que l'application nécessite certaines des fonctions trigonométriques pour les types à virgule flottante définis par l'application. Par conséquent, l'application instancie le paquetage générique Ada.Numerics.Generic\_Elementary\_Functions.

Les runtimes ZFP et Ravenscar-SFP n'incluent pas les bibliothèques mathématiques car ces runtimes sont destinés à être utilisés dans des systèmes certifiés. Une bibliothèque mathématique peut ne pas être nécessaire et dans ce domaine, la dépense de certification de l'implémentation de la bibliothèque inutilisée ne serait pas justifiée. Cependant, GNAT dispose d'implémentations de bibliothèques mathématiques certifiables incluses dans les run-times Ravenscar-Full. Par conséquent, nous copierons les fichiers des paquets mathématiques requis par l'application, ainsi que leurs dépendances, d'un runtime Ravenscar-Full dans le sous-répertoire gnat\_user de notre runtime.

La localisation des fichiers sources correspondant à un nom d'unité peut s'avérer quelque peu difficile au début, car les fichiers sources des run-time sont nommés dans le format codé "8.3". Les noms de fichier n'utilisent pas les noms d'unité complets, contrairement au schéma de dénomination par défaut utilisé pour les sources d'application, comme vous le verrez lorsque nous commencerons à copier tous les fichiers requis.

Nous allons d'abord copier la spécification et le corps du paquetage générique Ada.Numerics.Generic\_Elementary\_Functions, situés dans les fichiers a-ngelfu.ads et a-ngelfu.adb, où "a" signifie "Ada" et "ngelfu" est un encodage pour "Numerics.Generic\_Elementary\_Functions". Et bien sûr, nous avons besoin du paquetage parent Ada.Numerics, donc nous aurons ces fichiers copiés dans le sous-répertoire gnat\_user :

* a-ngelfu.ads
* a-ngelfu.adb
* a-numeri.ads

Ensuite, nous devons traverser la fermeture transitive des dépendances et copier également leurs sources. Si nous regardons le corps du paquet générique, nous voyons ce qui suit :

**with Ada.Numerics.Elementary\_Functions;with Ada.Numerics.Long\_Elementary\_Functions;with Ada.Numerics.Long\_Long\_Elementary\_Functions;**

**use ...**

**package body Ada.Numerics.Generic\_Elementary\_Functions is**

Par conséquent, nous avons besoin des sources de Ada.Numerics.Elementary\_Functions, Ada.Numerics.Long\_Elementary\_Functions, et Ada.Numerics.Long\_Long\_Elementary\_Functions. Les fichiers pour ces unités sont :

* a-nuelfu.ads (Ada.Numerics.Elementary\_Functions)
* a-nlelfu.ads (Ada.Numerics.Long\_Elementary\_Functions)
* a-nllefu.ads (Ada.Numerics.Long\_Long\_Elementary\_Functions)

Ces unités sont également définies par le langage afin qu'elles puissent être référencées par l'application, si nécessaire.

Les paquets n'ont pas de corps car chacun est une instanciation du paquet générique System.Generic\_C\_Math\_Interface. Nous devons également le copier :

* s-gcmain.ads
* s-gcmain.adb

Ces instanciations font également référence aux paquets libm de simple précision et de double précision System.Libm\_Single et System.Libm\_Double. Nous copions donc également ces fichiers :

* s-libsin.adb (System.Libm\_Single)
* s-libsin.adb
* s-libdou.adb (System.Libm\_Double)
* s-libdou.ads

Chacun d'eux fait référence au paquetage System.Libm\_Prefix, ainsi qu'à System.Libm, System.Libm\_Single.Squareroot et System.Libm\_Double.Squareroot :

* s-libpre.ads (System.Libm\_Prefix)
* s-libm.ads ((System.Libm)
* s-libm.adb
* s-lisisq.ads (System.Libm\_Single.Squareroot)
* s-lisisq.adb
* s-lidosq.ads (System.Libm\_Double.Squareroot)
* s-lidosq.adb

Ces paquets système de bas niveau sont compatibles avec notre nouvelle cible F429 car notre cible utilise la même configuration en virgule flottante (dans ce cas, le support matériel via le coprocesseur en virgule flottante).

L'ensemble des fichiers copiés est le suivant :

* a-ngelfu.ads (Ada.Numerics.Generic\_Elementary\_Functions)
* a-ngelfu.adb
* a-numeri.ads (Ada.Numerics)
* a-nuelfu.ads (Ada.Numerics.Elementary\_Functions)
* a-nlelfu.ads (Ada.Numerics.Long\_Elementary\_Functions)
* a-nllefu.ads (Ada.Numerics.Long\_Long\_Elementary\_Functions)
* s-gcmain.ads (System.Generic\_C\_Math\_Interface)
* s-gcmain.adb
* s-libsin.adb (System.Libm\_Single)
* s-libsin.ads
* s-libdou.adb (System.Libm\_Double)
* s-libdou.ads
* s-libpre.ads (System.Libm\_Prefix)
* s-libm.ads (System.Libm)
* s-libm.adb
* s-lisisq.ads (System.Libm\_Single.Squareroot)
* s-lisisq.adb
* s-lidosq.ads (System.Libm\_Double.Squareroot)
* s-lidosq.adb

Bien que cette liste soit assez longue, elle prend en charge la plupart des paquets mathématiques, et pas seulement ceux que nous voulions initialement, de sorte que l'ajout d'autres paquets mathématiques ne nécessiterait pas beaucoup d'ajouts supplémentaires.